

**Kazuhiro Ogata
Dominique Mery
Meng Sun
Shaoying Liu (Eds.)**

LNCS 15394

Formal Methods and Software Engineering

**25th International Conference
on Formal Engineering Methods, ICFEM 2024
Hiroshima, Japan, December 2–6, 2024
Proceedings**

 **Springer**

Lecture Notes in Computer Science

15394


Founding Editors

Gerhard Goos
Juris Hartmanis

Editorial Board Members

Elisa Bertino, *Purdue University, West Lafayette, IN, USA*

Wen Gao, *Peking University, Beijing, China*

Bernhard Steffen , *TU Dortmund University, Dortmund, Germany*

Moti Yung , *Columbia University, New York, NY, USA*

The series Lecture Notes in Computer Science (LNCS), including its subseries Lecture Notes in Artificial Intelligence (LNAI) and Lecture Notes in Bioinformatics (LNBI), has established itself as a medium for the publication of new developments in computer science and information technology research, teaching, and education.


LNCS enjoys close cooperation with the computer science R & D community, the series counts many renowned academics among its volume editors and paper authors, and collaborates with prestigious societies. Its mission is to serve this international community by providing an invaluable service, mainly focused on the publication of conference and workshop proceedings and postproceedings. LNCS commenced publication in 1973.

Kazuhiro Ogata · Dominique Mery · Meng Sun ·
Shaoying Liu
Editors


Formal Methods and Software Engineering


25th International Conference
on Formal Engineering Methods, ICFEM 2024
Hiroshima, Japan, December 2–6, 2024
Proceedings

Editors

Kazuhiro Ogata 
JAIST
Ishikawa, Japan

Meng Sun
Peking University
Beijing, China

Dominique Mery 
University of Lorraine
Nancy, France

Shaoying Liu 
Hiroshima University
Hiroshima, Japan

ISSN 0302-9743

ISSN 1611-3349 (electronic)

Lecture Notes in Computer Science

ISBN 978-981-96-0616-0

ISBN 978-981-96-0617-7 (eBook)

<https://doi.org/10.1007/978-981-96-0617-7>

© The Editor(s) (if applicable) and The Author(s), under exclusive license
to Springer Nature Singapore Pte Ltd. 2024

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

If disposing of this product, please recycle the paper.

Preface

The International Conference on Formal Engineering Methods (ICFEM) started in Hiroshima, Japan in 1997 and provided a forum for both researchers and practitioners to discuss and exchange their experience and results in research on theories, methods, languages, and supporting tools for integrating formal methods into conventional software engineering technologies to provide more effective and efficient approaches to large-scale software engineering. ICFEM 2024 was back to Hiroshima in Japan, the city of its birth, from December 2 to 6, 2024. At its first return to Hiroshima since 1997, ICFEM 2024 celebrated the 25th anniversary of the ICFEM conference series.

ICFEM 2024 received 50 submissions from 21 countries worldwide. The selection process was rigorous, with each paper receiving at least three Single-blind reviews. After thorough discussions, the program committee accepted 23 research papers. However, one paper was later withdrawn, resulting in 22 papers being included in the proceedings. The final acceptance rate was 44%. The accepted papers span a wide range of research areas, covering both theoretical foundations and practical applications of formal engineering methods.

We were honored to have the three distinguished keynote speakers Mike Hinchey, Naijun Zhan, and Mark Lawford, who shared their invaluable insights during their talks.

Mike Hinchey presented a work entitled *Formal Specification of Autonomy Features with ARE and KnowLang*, co-authored with Emil Vassev; both researchers are from Lero, the Science Foundation of Ireland Research Centre for Software Department of Computer Science and Information Systems, University of Limerick, Limerick, Ireland. Autonomous systems, such as autonomous vehicles, extend regular software-intensive systems with special autonomy features upstream. The identification of such features is not necessarily an easy task. Sometimes, they can be explicitly stated by stakeholders or in preliminary material available to requirements engineers. Often though, they are implicit, so a process of formal specification intended to capture the autonomy features has to be undertaken. The speaker elaborated on a methodology for capturing and specification of autonomy features where autonomy requirements are captured with ARE (Autonomy Requirements Engineering) and then are specified with KnowLang, a framework for knowledge representation and reasoning. In this approach, autonomy features are detected as special self-* objectives backed up by different capabilities and quality characteristics. The self-* objectives provide the system's ability to autonomously discover, diagnose, and cope with various problems. The captured autonomy requirements are formally specified with the KnowLang notation and then compiled to a knowledge base that is to be used by the KnowLang Reasoner.

Naijun Zhan (School of Computer Science, Peking University, Beijing, China) presented a work entitled *Synthesizing (Differential) Invariants by Reducing Non-Convex Programming to SDP*. Hybrid systems are integrations of discrete computation and continuous physical evolution. To guarantee the correctness of hybrid systems, formal techniques on modelling and verification of hybrid systems have been proposed. Hybrid CSP

(HCSP) is an extension of CSP with differential equations and some forms of interruptions for modelling hybrid systems, and Hybrid Hoare logic (HHL) is an extension of Hoare logic for specifying and verifying hybrid systems that are modelled using HCSP. The speaker reported an improved *HHL prover*, which is an interactive theorem prover based on Isabelle/HOL for verifying HCSP models. Compared with the prototypical release, the new HHL prover realises the proof system of HHL as a shallow embedding in Isabelle/HOL, rather than a deep embedding. In order to contrast the new HHL prover in shallow embedding and the old one in deep embedding, the speaker demonstrated the use of both variants on the safety verification of a lunar lander case study.

Mark Lawford (Department of Computing and Software, McMaster University, Canada) gave a talk entitled *Challenges and Opportunities in Assurance of Software Defined Vehicle*. Automotive innovation is increasingly software driven. As a result of the competitive environment requiring a yearly release of new automotive models featuring the latest driver assistance features and innovation in electrification, safety critical software is being developed at an unprecedented scale on extremely tight deadlines. To be competitive automotive manufacturers and parts suppliers need to change how they develop and assure software intensive systems. This presents a tremendous opportunity for researchers in the application of formal methods and model-based software engineering to have a significant impact on industry practice. This talk highlighted recent work on model-based software engineering, incremental assurance, and the potential for applications of formal methods to help the automotive industry make the transition to the Software-Defined vehicle.

The EasyChair conference management system was set up for ICFEM 2024, supporting submissions, reviewing, and volume-editing processes. We acknowledge that it is an outstanding tool for the academic community. We would like to thank all the authors who submitted their work to ICFEM 2024. We are grateful to the program committee members and external reviewers for their high-quality reviews and discussions. Finally, we wish to thank the Organizing Committee members for their hard work and continuous support. We would also like to thank Springer and the publishing team managed by Ronan Nugent for their continuous support and assistance in producing the conference proceedings. Finally, we extend our sincere gratitude to our sponsors Murata Science and Education Foundation and Huawei Technologies Co., Ltd. and the supporters Information Processing Society of Japan, Hiroshima University, and IEEE Japan Council.

We hope that the papers in these proceedings will engage readers and inspire new ideas for future research.

September 2024

Kazuhiro Ogata
Dominique Mery
Meng Sun
Shaoying Liu

Organization

General Chair

Shaoying Liu

Hiroshima University, Japan

Program Chairs

Kazuhiro Ogata

JAIST, Japan

Meng Sun

Peking University, China

Dominique Méry

LORIA, Telecom Nancy, University of Lorraine,
France

Program Committee

Yamine Aït-Ameur

IRIT/INPT-ENSEEIH, France

Étienne André

Université Sorbonne Paris Nord, France

Cyrille Valentin Artho

KTH Royal Institute of Technology, Sweden

Guangdong Bai

University of Queensland, Australia

Christel Baier

TU Dresden, Germany

Lei Bu

Nanjing University, China

Ana Cavalcanti

University of York, UK

Yean-Ru Chen

National Cheng Kung University, Taiwan

Yuting Chen

Shanghai Jiao Tong University, China

Zhenbang Chen

National University of Defense Technology,
Changsha, China

Ranald Clouston

Aarhus University, Denmark

Florin Craciun

Babeş-Bolyai University, Romania

Ana De Melo

University of São Paulo, Brazil

Duong Dinh Tran

JAIST, Japan

Thi Thu Ha Doan

Freiburg University, Germany

Naipeng Dong

National University of Singapore, Singapore

Aaron Dutle

NASA, USA

Santiago Escobar

Universitat Politècnica de València, Spain

Flavio Ferrarotti

Software Competence Centre Hagenberg, Austria

Marc Frappier

Université de Sherbrooke, Canada

Daniel Gaina

Kyushu University, Japan

Lindsay Groves	Victoria University of Wellington, New Zealand
Osman Hasan	National University of Sciences and Technology, Pakistan
Xudong He	Florida International University, USA
Zhe Hou	Griffith University, Australia
Fuyuki Ishikawa	National Institute of Informatics, Japan
Eun-Young Kang	University of Southern Denmark, Denmark
Mark Lawford	McMaster University, Canada
Jiaying Li	Microsoft, China
Shang-Wei Lin	Nanyang Technological University, Singapore
Guanjun Liu	Tongji University, China
Shaoying Liu (Chair)	Hiroshima University, China
Si Liu	ETH Zurich, Switzerland
Zhiming Liu	Southwest University, China
Frederic Mallet	Côte d'Azur University, France
Heiko Mantel	TU Darmstadt, Germany
Diego Marmsoler	University of Exeter, UK
Narciso Marti-Oliet	Universidad Complutense de Madrid, Spain
Dominique Mery	Université de Lorraine, LORIA, France
Stephan Merz	Inria Nancy, France
Canh Minh Do	JAIST, Japan
Rosemary Monahan	Maynooth University, Ireland
Shin Nakajima	National Institute of Informatics, Japan
Masaki Nakamura	Toyama Prefectural University, Japan
Kazuhiro Ogata (Chair)	JAIST, Japan
Jun Pang	University of Luxembourg, Luxembourg
Yu Pei	Hong Kong Polytechnic University, China
Shengchao Qin	Teesside University, UK
Silvio Ranise	University of Trento and Fondazione Bruno Kessler, Trento, Italy
Elvinia Riccobene	University of Milan, Italy
Adrian Riesco	Universidad Complutense de Madrid, Spain
Markus Roggenbach	Swansea University, UK
Subhajit Roy	Indian Institute of Technology Kanpur, India
Rubén Rubio	Universidad Complutense de Madrid, Spain
David Sanan	Singapore Institute of Technology, Singapore
Jing Sun	University of Auckland, New Zealand
Meng Sun	Peking University, China
Sofiene Tahar	Concordia University, Canada
Maurice ter Beek	CNR, Italy
Cong Tian	Xidian University, China
Elena Troubitsyna	KTH, Sweden

Tatsuhiko Tsuchiya	Osaka University, Japan
Bow-Yaw Wang	Academia Sinica, Taiwan
Hai H. Wang	University of Aston, UK
Tomoyuki Yokogawa	Okayama Prefectural University, Japan
Min Zhang	East China Normal University, China
Xiyue Zhang	University of Oxford, UK
Yongwang Zhao	Zhejiang University, China
Peter Ölveczky	University of Oslo, Norway
Ionuț Țuțu	Institute of Mathematics of the Romanian Academy, Romania

Additional Reviewers

Ahmed, Asad	Luan, Xiaokun
Ashraf, Sobia	Morelli, Umberto
Barhoumi, Oumaima	Nguyen, Hoang Nga
Berlato, Stefano	Tiwari, Mukesh
Bu, Hao	Wang, Chao
Bukhari, Syed Ali Asadullah	Wei, Ran
Deniz, Elif	Yang, Min
Graydon, Mallory	Yin, Zijing
He, Leifeng	Zhang, Yuanrui

Contents

NL2CTL: Automatic Generation of Formal Requirements Specifications via Large Language Models	1
<i>Mengyan Zhao, Ran Tao, Yanhong Huang, Jianqi Shi, Shengchao Qin, and Yang Yang</i>	
Repairing Event-B Models Through Quantifier Elimination	18
<i>Tsutomu Kobayashi and Fuyuki Ishikawa</i>	
Tuning Trains Speed in Railway Scheduling	37
<i>Étienne André</i>	
The Bright Side of Timed Opacity	51
<i>Étienne André, Sarah Dépernet, and Engel Lefaucheux</i>	
Clock-Dependent Probabilistic Timed Automata with One Clock and No Memory	70
<i>Jeremy Sproston</i>	
Efficient State Estimation of Discrete-Timed Automata	85
<i>Julian Klein, Paul Kogel, and Sabine Glesner</i>	
LRNN: A Formal Logic Rules-Based Neural Network for Software Defect Prediction	106
<i>Yuxiang Shang and Shaoying Liu</i>	
Quantitative Symbolic Robustness Verification for Quantized Neural Networks	125
<i>Mara Downing, William Eiers, Erin DeLong, Anushka Lodha, Brian Ozawa Burns, Ismet Burak Kadron, and Teyfik Bultan</i>	
Graph Convolutional Network Robustness Verification Algorithm Based on Dual Approximation	146
<i>Dongdong An, Hao Zhang, Qin Zhao, Jing Liu, Jianqi Shi, Yanhong Huang, Yang Yang, Xu Liu, and Shengchao Qin</i>	
Formal Kinematic Analysis of Epicyclic Bevel Gear Trains	162
<i>Kubra Aksoy, Adnan Rashid, and Sofiène Tahar</i>	
Deciding the Synthesis Problem for Hybrid Games Through Bisimulation	181
<i>Catalin Dima, Mariem Hammami, Youssouf Oualhadj, and Régine Laleau</i>	

Formal Analysis of FreeRTOS Scheduler on ARM Cortex-M4 Cores	199
<i>Chen-Kai Lin and Bow-Yaw Wang</i>	
Differential Property Monitoring for Backdoor Detection	216
<i>Otto Brechelmacher, Dejan Ničković, Tobias Nießen, Sarah Sallinger, and Georg Weissenbacher</i>	
MemSpate: Memory Usage Protocol Guided Fuzzing	237
<i>Zhiyuan Fu, Jiacheng Jiang, Cheng Wen, Zhiwu Xu, and Shengchao Qin</i>	
The Continuum Hypothesis Implies the Existence of Non-principal Arithmetical Ultrafilters – A Coq Formal Verification	257
<i>Guowei Dou, Si Chen, Wensheng Yu, and Ru Zhang</i>	
Observability of Boolean Control Networks: New Definition and Verification Algorithm	278
<i>Guisen Wu, Zhiming Liu, and Jun Pang</i>	
Formalizing Potential Flows Using the HOL Light Theorem Prover	297
<i>Elif Deniz and Sofiene Tahar</i>	
On-the-Fly Proof-Based Verification of Reachability in Autonomous Vehicle Controllers Relying on Goal-Aware RSS	314
<i>Peter Rivière, Tsutomu Kobayashi, Neeraj Kumar Singh, Fuyuki Ishikawa, Yamine Ait Ameur, and Guillaume Dupont</i>	
Efficient SMT-Based Model Checking for HyperTWTL	332
<i>Ernest Bonnah, Luan Viet Nguyen, and Khaza Anuarul Hoque</i>	
A Tableau-Based Approach to Model Checking Linear Temporal Properties ...	353
<i>Canh Minh Do, Tsubasa Takagi, and Kazuhiro Ogata</i>	
Simple LTL Model Checking on Finite and Infinite Traces over Concrete Domains	375
<i>David Doose and Julien Brunel</i>	
Model Checking Concurrency in Smart Contracts with a Case Study of Safe Remote Purchase	391
<i>Yisong Yu, Naipeng Dong, Zhe Hou, and Jin Song Dong</i>	
Author Index	409



NL2CTL: Automatic Generation of Formal Requirements Specifications via Large Language Models

Mengyan Zhao^{1,2,5}, Ran Tao^{1,2,5}, Yanhong Huang^{1,2,5}(✉), Jianqi Shi^{1,2,5}, Shengchao Qin^{3,4}, and Yang Yang^{1,2,5}

- ¹ National Trusted Embedded Software Engineering Technology Research Center, East China Normal University, Shanghai, China
² Hardware/Software Co-Design Technology and Application Engineering Research Center, East China Normal University, Shanghai, China
³ Guangzhou Institute of Technology, Xidian University, Xi'an, China
⁴ ICTT and ISN Laboratory, Xidian University, Xi'an, China
⁵ Software Engineering Institute, East China Normal University, Shanghai, China
yhuang@sei.ecnu.edu.cn

Abstract. Reducing the gap between natural language requirements and precise formal specifications is a critical task in requirements engineering. In recent years, requirement engineering is becoming increasingly complex alongside the growing intricacy of system engineering. Most requirements are expressed in natural language, which can be incomplete and ambiguous. However, formal languages with strict semantics can accurately represent certain temporal logic properties and allow for automated verification and analysis. This often limits the application of verification techniques, as writing formal specifications is a manual, error-prone, and time-consuming task. To address this, this paper proposes a framework that leverages Large Language Models (LLMs) to achieve automated conversion of natural language requirements to Computation Tree Logic (CTL). To address the issue of dataset scarcity, we leveraged the interactive and generative capabilities of LLMs. By constructing a random generation algorithm and utilizing prompt engineering, we generated an NL-CTL dataset using LLMs. The generated dataset was then used to fine-tune the T5-Large model, enhancing its generative capacity. To improve generalization, this paper proposes the use of the GPT-3.5 Atomic Proposition (AP) Recognition method, which eliminates the constraints of using the framework across different domains. A series of experimental evaluations showed that the fine-tuned LLM achieved an accuracy of 46.4%, whereas the LLM with few-shot learning using only prompt engineering achieved only 2% accuracy, demonstrating the feasibility of this approach.

Keywords: Requirements Engineering (RE) · Specification Generation · Computation Tree Logic (CTL) · Large Language Models (LLM)

1 Introduction

As the scale and complexity of software continue to increase, software reliability issues have gained significant attention. Requirements engineering (RE), being the first phase of the software life cycle, is a crucial part of software engineering. Efficient management of RE can accelerate the software development process [21]. However, in practical software development, most software requirements are written in natural language. The inherent ambiguity and imprecision of natural language make it challenging to effectively ensure the completeness, consistency, correctness, and reliability of software systems, thereby increasing the difficulty of analyzing and processing requirements [29]. Formal specifications are mathematical descriptions of system properties, behaviors, and constraints, used in fields such as system design, requirements analysis, and automated reasoning in computer science. Compared to natural language (NL), formal specifications can precisely define system behavior, eliminate ambiguity, and support the design and verification of system software. Due to the need for substantial domain-specific knowledge and a significant amount of manual work, the application of formal specification languages is still almost exclusively performed by domain experts.

The difficulty of using a formal specification language leads to the gap between its advantages in software development and actual practice, so the direct conversion of natural language to formal language is necessary for the practical application of formal specification language. Writing formal specifications manually is not only a threshold for writers, but also very time-consuming and error-prone, especially when dealing with complex systems. To address this, researchers have introduced numerous automatic and semi-automatic methods for formalizing requirements, such as template-based and deep learning methods for generating formal specifications. Both methods do speed up the process of writing formal specifications and reduce the probability of human error. However, template-based generation of formal specifications has poor flexibility and users can only choose according to the template language. Formal specifications generated based on deep learning may be more dependent on the statistical patterns in the training data, and thus to a certain extent there may be linguistic styles or patterns related to the training data. These methods have certain limitations, including the need for extensive manual construction and maintenance work, as well as difficulties in adapting to different environments [30]. However, with the latest advances in Natural Language Processing (NLP) technology, particularly the development and application of generative Large Language Models (LLMs), it has become possible to overcome these limitations [2]. LLMs may be more accurate and coherent when generating formal specifications due to their pre-training approach which gives it better language understanding and generation capabilities, and thus the generated specifications may be more accurate, coherent, and better able to understand input natural language.

This paper uses LLMs to automatically generate formal specifications in Computation Tree Logic (CTL) based on natural language requirements, addressing the high specialization demands of formal verification methods. CTL,

a branching-time logic, offers higher computational efficiency in model checking, and many industrial model checking tools use CTL as the specification language. A significant bottleneck in NL-to-CTL research is the lack of data. Although modern statistical methods can surpass rule-based approaches [4], they typically require large datasets, which are costly and difficult to collect, necessitating highly specialized annotators [3]. To supplement the data creation process and mitigate the need for large datasets, this paper employs pre-trained LLMs. Specifically, we use GPT-3.5 to assist in dataset creation and fine-tune the T5-Large model [27] to achieve the conversion from NL to CTL. To validate the usability of this method, we set up a series of evaluation experiments. By training and testing the LLM on generated NL-CTL datasets of varying scales, we revealed that the more training data used, the higher the accuracy of the fine-tuned model. Additionally, when comparing the fine-tuned LLM with the baseline prompted GPT-3.5, the experimental result shows that the fine-tuned LLM achieved an accuracy of 46.4%, whereas the LLM using only prompt engineering for few-shot learning achieved only 2% accuracy, demonstrating the feasibility of the proposed method.

Our main contributions are as follows:

1. **Construct a cross-domain NL-CTL dataset.** To address the lack of datasets, we leveraged the interaction and generation capabilities of LLMs. By developing a random generation algorithm and prompt engineering, we used an innovative GPT-3.5-assisted framework to generate a dataset of 3K enhanced NL-CTL pairs. Additionally, we enhanced the generality of the data by using the “lifted” version of NL and CTL, where all atomic propositions (APs) in the data were hidden.
2. **Fine-tuning the lifted NL-to-CTL model.** We fine-tuned the T5-Large model using the constructed dataset to improve the generative capabilities of the large model. To enhance generalization, we used the GPT-3.5 AP Recognition method to eliminate the constraints of using the framework across different domains. The experimental result showed that the fine-tuned T5-Large model achieved higher accuracy compared to the baseline prompted GPT-3.5, demonstrating the feasibility of the proposed method.

This paper is organized as follows. Section 2 gives an overview of LLMs and CTL semantics and discusses related work. Section 3 describes the framework and algorithm of our approach. In Sect. 4, we introduce the experimental setup and discuss the experimental evaluation result. Section 5 concludes our work.

2 Background and Related Work

2.1 Large Language Models

Large Language Models (LLMs) are neural network models trained using massive text data based on deep learning techniques. They are capable of understanding and generating natural language text and perform well in a variety of linguistic

tasks, including translation, text generation, summarization, Q&A and dialogue systems. Common LLMs include OpenAI’s GPT series, Google’s BERT and T5. Among them, GPT-1 explores the natural language task solving capability of decoder-only Transformer architecture under the “pre-training + fine-tuning” paradigm; GPT-2 preliminarily verifies the effectiveness of scaling up the parameters of the model (scaling up law), and explores the natural language cue-based LLMs. GPT-3 explores the effect of language models with hundreds of billions of parameters for the first time, and proposes a task solving method based on “context learning”; The CodeX [5] uses code data to fine-tune GPT-3 to improve code ability and complex reasoning; InstructGPT [25] uses reinforcement learning based on human feedback (RLHF) to strengthen the ability to follow human commands and align human preferences; ChatGPT is similar to InstructGPT, but further introduces dialog data for learning, thus strengthening the ability to have multi-round conversations; GPT-3.5 [1] is able to handle a longer context window, and can be used to solve tasks with “contextual learning”; CodeX [5] uses code data to fine-tune GPT-3 to improve coding ability and complex reasoning. Handle longer context windows, has multimodal comprehension capability, and has significantly improved capabilities in logical reasoning and complex task processing.

LLM techniques mainly include model pre-training, setup fine-tuning, cue learning, knowledge enhancement and tool learning. Natural Language to Programming Language, Codex [5] model is a GPT language model fine-tuned based on GitHub public code, which is capable of generating corresponding code based on natural language instructions. Natural Language to Formal Specification, nl2spec [11] is a framework for applying LLMs to derive formal specifications (in temporal logic) from unstructured natural language.

2.2 Computation Tree Logic

Computation Tree Logic (CTL) is a type of branching-time logic, meaning that its time model is a tree-like structure where the future is uncertain; there are multiple paths in the future, any of which could be the actual path that is realized. CTL belongs to a class of temporal logics that includes Linear Temporal Logic (LTL). LTL provides an intuitive and precise mathematical notation for expressing linear-time properties. LTL formulas are suitable for describing requirements with both logical and temporal properties, allowing for automated verification. The syntax of LTL can be recursively defined over a set of atomic propositions AP as follows:

$$\varphi ::= \text{true} \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \cup \varphi_2 \quad (1)$$

where $a \in AP$ is an atomic proposition, φ is an LTL formula. \neg and \wedge represent propositional logic negation and conjunction, respectively. \bigcirc is the temporal operator next, and \cup is the temporal operator until. The formula $\bigcirc\varphi$ represents that φ is true in the next state. $\varphi_1 \cup \varphi_2$ indicates that φ_2 is true in some state, and φ_1 is always true in all preceding states. Additionally, disjunction \vee , implication

\rightarrow , eventually \diamond , and always \square can be derived from the aforementioned operators as derived operators. Among them, \bigcirc , \cup , \diamond , and \square are temporal operators. LTL can only be verified along a single timeline and cannot distinguish temporal behaviors on different paths. This limitation becomes evident when dealing with branching systems, such as concurrent systems and distributed systems.

CTL addresses this issue by introducing path quantifiers, allowing for the description of temporal behaviors across different paths. It describes the properties of system states as they change over time by combining path quantifiers with temporal operators. The syntax of CTL is as follows:

$$\phi ::= \text{true} \mid a \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \forall\varphi \mid \exists\varphi \quad (2)$$

$$\varphi ::= \bigcirc\phi \mid \phi_1 \cup \phi_2 \mid \diamond\phi \mid \square\phi \quad (3)$$

where ϕ represents a state formula, while φ represents a path formula, indicating a path starting from the root node of the tree. \forall is the universal quantifier, indicating all future paths. \exists is the existential quantifier, indicating the existence of at least one future path. From these two formulas, it can be observed that in CTL formulas, temporal operators are always preceded by path quantifiers.

2.3 Related Work

For decades, researchers have developed methods to translate natural language sentences into various target language formulas [3, 13, 28]. In recent years, how to obtain better automated tool support for efficiently converting natural language requirements into formal specifications has become a research hotspot in the field of formal methods. A significant amount of work in this new phase has utilized interactive training [33, 37] and physical demonstrations to infer task constraints [7, 9, 31] and LTL formulas [8, 10, 32, 35]. Early work on translating natural language into formal specification focused on grammar-based approaches [16, 23], which can handle structured natural language. Additionally, there are interactive methods that use SMT solving and semantic parsing [15], as well as structured temporal methods rooted in robotics [36] and planning [26]. However, to simplify tasks, previous works have often made strong assumptions to constrain the input text or the output formulas, thereby limiting flexibility and generality. For example, Finucane et al. [13], Taylor et al. [34], and Howard et al. [19] all use the traditional approach of preprocessing the given English input by restricting the input NL and extracting syntactic information, then identifying the patterns or rules of the TL through classification and running an attribute-based grammar parser to derive the target logical format.

With the rise of ChatGPT, neural networks have also been used for the generation of formal specifications in the past two years. For example, methods include training STL models from scratch [18], fine-tuning language models [17], or applying GPT-3 in a one-shot manner [14, 24]. Cosler *et al.* [11] used LLMs to map formal sub-formulas back to the corresponding natural language segments in the input, aiming to detect and resolve the inherent ambiguities in natural language system requirements. However, this method still requires human intervention to interactively add, edit, and delete sub-translations to improve accuracy.

Chen *et al.* [6] proposed an accurate and generalizable framework for converting English instructions from NL to TL by constructing an NL-TL dataset using LLMs and fine-tuning the LLM to enhance the accuracy of formal specification generation. Due to the lack of datasets across different application domains and the inherent complexity of CTL itself, the translation between natural language and CTL has not been adequately studied. Therefore, this paper proposes a methodological framework for the automatic generation of CTL formulas from natural language requirements using LLMs.

3 Methodology

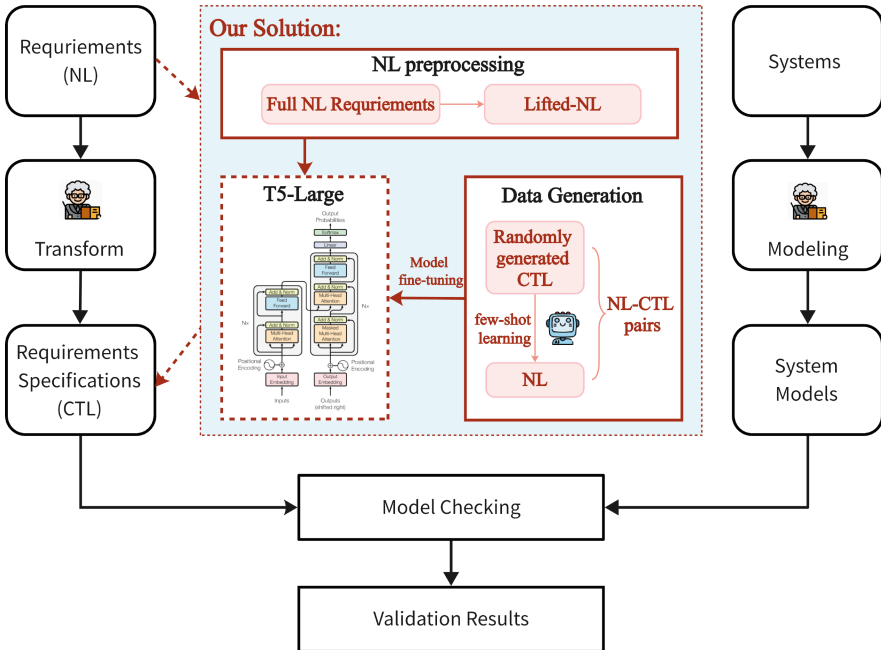


Fig. 1. The workflow of our study.

Figure 1 shows the workflow of formal verification in requirements engineering with black arrows. In this workflow, experts convert natural language requirements into formal specifications using template-based or manual methods [12, 22] and model the system using various formal models. Finally, model checking is performed on both the requirement specifications and the system model to verify the correctness of the requirements. Our approach leverages LLMs to achieve the conversion of natural language requirements to CTL. First, natural language requirements are pre-processed and converted into lifted NL to facilitate LLM

understanding. These are then input to the fine-tuned LLM to generate corresponding CTL formulas. The fine-tuning of the T5-Large model involves three steps: first, constructing a random algorithm to generate a series of CTL formulas; second, using few-shot learning methods with LLMs to convert these into lifted NL to enrich the dataset; and third, fine-tuning the LLM with the generated NL-CTL pairs to improve model accuracy.

3.1 Lifted NL and Lifted CTL

Hsiung et al. [20] proposed a new LTL nomenclature called ‘lifted’ LTL, which hides specific atomic propositions (APs) corresponding to individual operations. In this approach, each AP is replaced with a placeholder *prop.i*. To enhance the generality of model, we represent data as ‘lifted’ NL and CTL. This allows our model to be trained on the general context of instructions, regardless of specific APs. The correspondence between full NL/CTL and lifted NL/CTL is shown in Fig. 2.

In previous work, models trained for NL to TL (Temporal Logic) conversion typically involved translating specific actions into APs. For example, the AP “Create a response in Slack” could be formalized as “CreateSlack”. This necessitates each work to standardize its own AP content and style, thereby affecting generalization. In this paper, instead, we use the CPT-3 AP Recognition method to hide all APs in the data during fine-tuning and achieve a lifted model for the conversion from lifted NL to CTL. That is, we used LLMs to recognize APs in natural sentences, such as “create a response in Slack”, and instead of translating it to “create.Slack”, it is masked as a placeholder *prop.i*.

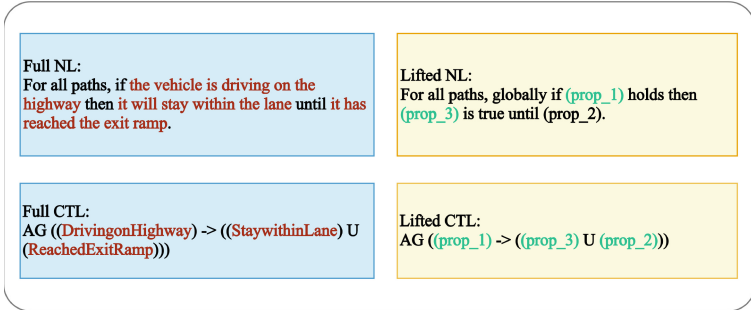


Fig. 2. Illustration of lifted NL and lifted CTL.

3.2 Data Generation

This paper utilizes the LLM GPT-3.5 to aid in generating lifted NL and CTL pairs to construct the model fine-tuning dataset. Previous work primarily adopted an intuitive approach, using prompt engineering to generate additional NL-CTL pairs through few-shot learning with various NL-CTL pairs as

prompts. However, it was found that under this approach, the model consistently generated NL and CTL with syntactic structures similar to the given prompts, thereby limiting the diversity of the data. To encourage the model to produce a wider variety of sentences, we instructed it to generate corresponding NL from different CTLs.

We used a binary tree generation algorithm to randomly synthesize various pre-order CTLs, which are then converted into ordered expressions according to specific rules. Algorithm 1 illustrates the method for randomly synthesizing various pre-order CTLs, where the input is the maximum number of APs N and the output is a pre-order CTL *pre_orderCTL*. All operators are classified into those with only *one_leaves* and those with *two_leaves*. First, a random integer value within $(1, N)$ is obtained as the total number of APs, then a randomly ordered prop list of length *AP_nums* is generated and divided into several sub-lists *sub_lists* (Line 4–7). For each sub-list, operators are randomly appended to the left until each prop occupies a position in the binary tree (Line 9–12). Finally, by attaching operators with *two_leaves*, these modified sub-lists are assembled back into a complete CTL (Line 13).

Algorithm 1. Algorithm for randomly generating pre-order CTL.

```

1: Input:  $N$ : Maximum number of APs
2: Output: pre_orderCTL: Synthesized pre-order CTL
3:
4: two_leaves = [ $\&$ ,  $\rightarrow$ ,  $\leftrightarrow$ ,  $|$ ,  $U$ ];
5: one_leaves = [ $\neg$ , AG, EG, AF, EF, AX, EX, A, E];
6: AP_nums = Random.randint(1,  $N$ );
7: sub_lists  $\leftarrow$  getRandomCombinationAP(AP_num);
8:
9: repeat
10:   sub_list  $\leftarrow$  insertRandomOperators(two_leaves, one_leaves);
11:   sub_CTL  $\leftarrow$  sub_list
12: until sub_lists =  $\emptyset$ 
13: pre_orderCTL  $\leftarrow$  getPreOrderCTL(sub_CTL, two_leaves);

```

To make the input CTL more understandable to GPT-3.5, operators were represented by words indicating their meanings (e.g., \Rightarrow (implies), \Leftrightarrow (equivalent), \vee (or), etc.). GPT-3.5 then attempted to generate original NL sentences that closely match the semantics of the CTL. During this process, the NL-CTL pairs in the prompts were carefully selected to enhance lexical and structural diversity. We collected 200 NL instructions from 10 volunteers familiar with robotic tasks and randomly selected 100 NLs as a prompt pool and another 100 NLs as manual test data. In each iteration, 20 pairs were randomly chosen from the prompt pool to serve as prompts for GPT-3.5, with examples of prompts shown in Fig. 3.

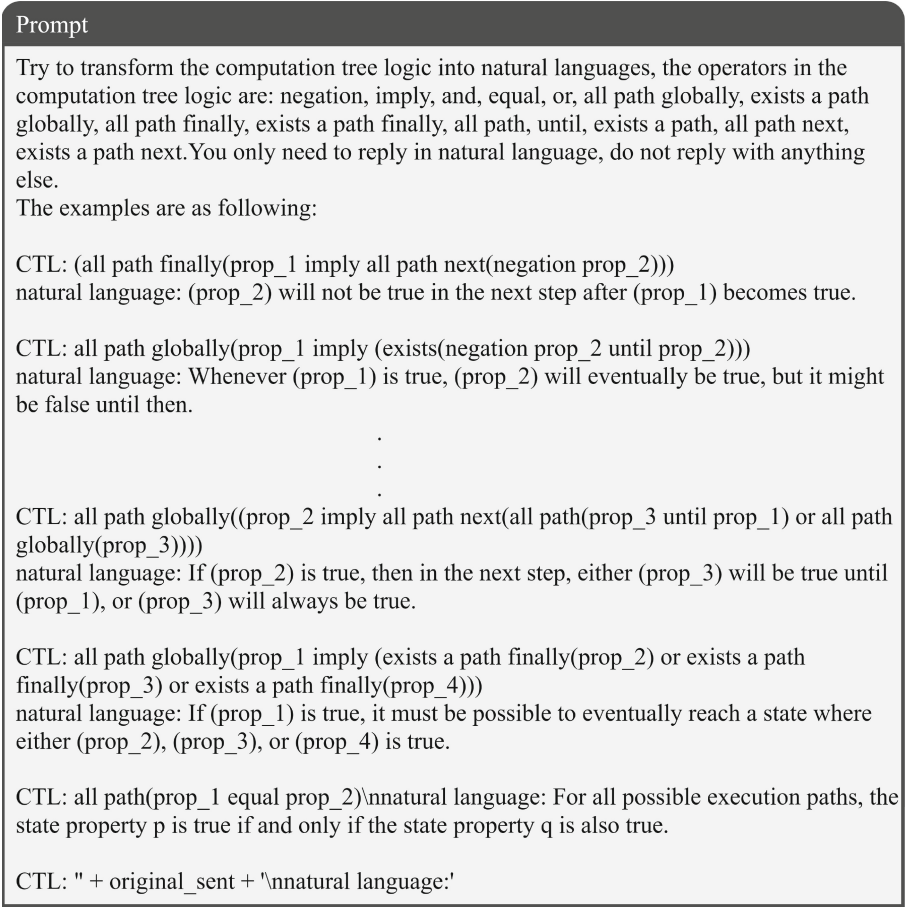


Fig. 3. Prompts for converting from synthesized CTL to NL via GPT-3.5.

3.3 Model Fine-Tuning

After constructing the dataset, it is used to fine-tune the LLM to enhance the accuracy of NL2CTL conversion. The T5 model [27], proposed by the Google Brain team, is a sequence-to-sequence (Seq2Seq) model based on the Transformer architecture. Its primary feature is converting various NLP tasks (such as translation, summarization, and question answering) into a unified framework for training, using a text-to-text unified model paradigm, which ensures model flexibility. The T5 model employs mixed-precision training and adaptive optimizers to accelerate the training process, and it utilizes data filtering and dynamic batching to improve data efficiency, boasting excellent generalization and transfer capabilities. T5-Large, a variant within the T5 family, has approximately 770 million parameters, making it a medium-sized pre-trained model. In

this paper, the T5-Large (770M) model is chosen as the base LLM for model fine-tuning, with experimental settings described in Sect. 4.

4 Implementation

4.1 Experimental Setup

The paper focuses on the feasibility of converting NL to CTL via LLM, and the experiment is focused on enhancing the NL2CTL capability of the LLM by fine-tuning it. The purpose of the experiment is to demonstrate the feasibility of implementing NL2CTL using a LLM, and to show the effect of the size of the training dataset on the ability of the LLM to NL2CTL. The experiment consists of three main approaches: Improved NL-CTL transition performance by fine-tuning the T5-large model as shown in Fig. 4; Hide APs as “prop_i”, anonymize them with the “prop_i” placeholder to maintain the generality of the framework; Generate NL-CTL datasets by prompting GPT-3.5 to direct it to generate datasets suitable for fine-tuning the T5-large model as shown in Fig. 5.

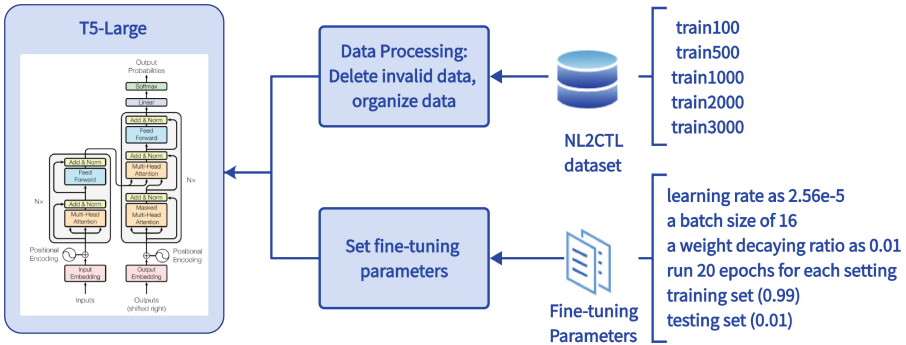


Fig. 4. Fine-tuning the T5-Large Model.

The LLM we chose for fine-tuning was T5-large, which was trained separately using five datasets with different amounts of data (containing 100, 500, 1000, 2000 and 3000 pieces of data, respectively). For all the fine-tuning experiments on T5-large model, we choose the learning rate as $2.56e-5$, a batch size of 16, a weight decaying ratio as 0.01, and run 20 epochs for each setting. Training and testing is performed on a single RTX 2080 Ti x2 (22GB) GPU. For the finetuning on lifted models, the input dataset is split into a training set (0.99) and a testing set (0.01). Finally, all fine-tuned models are tested on the same dataset (containing 500 data points), and to ensure the accuracy of the results, we perform three rounds of testing for each fine-tuned models, and eventually take the average of the test accuracies.

In practical applications, we need to structure the APs in CTL (such as “verb.noun”) to allow for direct connection with controllers. Then We use GPT-3

to identify APs in the sentence and mask them as “prop_i”. APs are anonymized using placeholders similar to “prop_i” to maintain generality and simplify integration with control systems. In the future, if there are fields to use the NL2CTL framework proposed in this paper, it can be combined with AP recognition task.

The T5-Large model excels in multi-task learning and unified text-to-text methods for a wide range of NLP tasks. GPT-3.5, on the other hand, has advantages in complex task processing and generation capabilities and is more suitable for demanding application scenarios. The T5-Large model is categorized as a compact model, in contrast to the GPT-3.5, which is recognized as a substantially expansive model within the spectrum of large-scale models. The experiments use GPT-3.5’s implementation of NL2CTL via an end-to-end approach as the baseline for the experiments, i.e. GPT-3.5 directly converts NL to CTL with a small amount of learning and cueing, and the same is tested using the same dataset for the learned cued GPT-3.5 model. The experiment stores the results of each tuned or learned model test in a table for easy manual review, and calculates the test accuracy for comparative analysis.

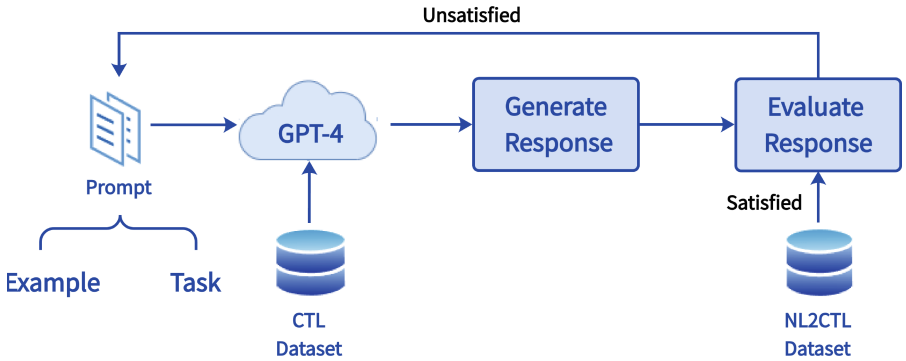


Fig. 5. NL2CTL Dataset Generation

4.2 DataSet

We did not find a suitable NL2CTL dataset in the currently publicly available dataset, and in order to allow fine-tuning in experiments as well as to test LLMs, we propose a method to let it generate a suitable dataset by prompt of GPT-3.5. Different datasets conforming to the CTL syntax were randomly generated by this method (training dataset with 100, 500, 1000, 2000, 3000 data and test datasets with 500 data). The process of generating the NL2CTL dataset is shown in Fig. 5.

4.3 Baseline

Prompt

Objective: You are a Computation Tree Logic (CTL) expert. Your answers always need to follow the following output format and you always have to try to provide a CTL formula. You may repeat your answers. End your answer with "So the final CTL translation is:" + the result obtained by translation. Please answer in text format only, do not use latex or markdown format.

Method: Translate the following natural language sentences into a CTL formula and explain your translation step by step. Remember that A means "for all paths", E means "there exists a path", X means "next", U means "until", G means "globally", F means "finally", which AG means "This is always true for all paths", and EF means "eventually exist".

The formula should only contain atomic propositions or operators $|$, $\&$, \sim , \rightarrow , \leftrightarrow , A, E, X, U, G, F. Atomic propositions are replaced by (prop_n)

Example:

Natural Language: For all paths, globally if (prop_1) holds then (prop_3) is true until (prop_2).

Given translations: {}

Explanation: "(prop_1) holds" from the input translates to the atomic proposition (prop_1).

"(prop_3) is true until (prop_2)" from the input translates to the subformula (prop_3) U (prop_2).

"if x then y" translates to an implication $x \rightarrow y$, so "if a holds then c is true until b" translates to an implication (prop_1) \rightarrow ((prop_3) U (prop_2)).

"Globally" from the input translates to the temporal operator G. "For all paths" translates to the path quantifier A.

Explanation dictionary: {"(prop_1) holds": "(prop_1)", "(prop_3) is true until (prop_2)": "(prop_3) U (prop_2)", "if (prop_1) holds then (prop_3) is true until (prop_2)": "(prop_1) \rightarrow ((prop_3) U (prop_2))", "Globally": "G", "For all paths": "A"}

So the final CTL translation is: A G ((prop_1) \rightarrow ((prop_3) U (prop_2))).FINISH

Natural Language: There exists a path where eventually request (prop_1) is followed by a grant (prop_2).

Given translations: {}

Explanation: "Request (prop_1)" from the input translates to the atomic proposition (prop_1) and "grant (prop_2)" translates to the atomic proposition (prop_2).

"Eventually" translates to the temporal operator F. "Followed by" is the natural language representation of an implication.

"There exists a path" translates to the path quantifier E.

Explanation dictionary: {"Request (prop_1)": "(prop_1)", "grant (prop_2)": "(prop_2)", "Eventually": "F", "Followed by": " \rightarrow ", "There exists a path": "E"}

So the final CTL translation is: E F ((prop_1) \rightarrow (prop_2)).FINISH

Fig. 6. Prompts for converting from synthesized NL to CTL via GPT-3.5.

To the best of our knowledge, there is no relevant research on the implementation of NL2CTL using LLMs, so we decided to use the test results of GPT-3.5 directly converting NL to CTL with few-shot learning as a baseline to compare with those of the fine-tuned T5-Large model in the experiments, and the results of the comparison are shown in Fig. 8. Generating the baseline: create a dialogue with GPT-3.5, add examples and appropriate guidelines to the prompt as in Fig. 6. GPT-3.5 learns from this conversation and then sends it the test data, records its responses and compares them with the CTL in the dataset, records the responses in a table, and calculates the test accuracy of the baseline (Fig. 7).

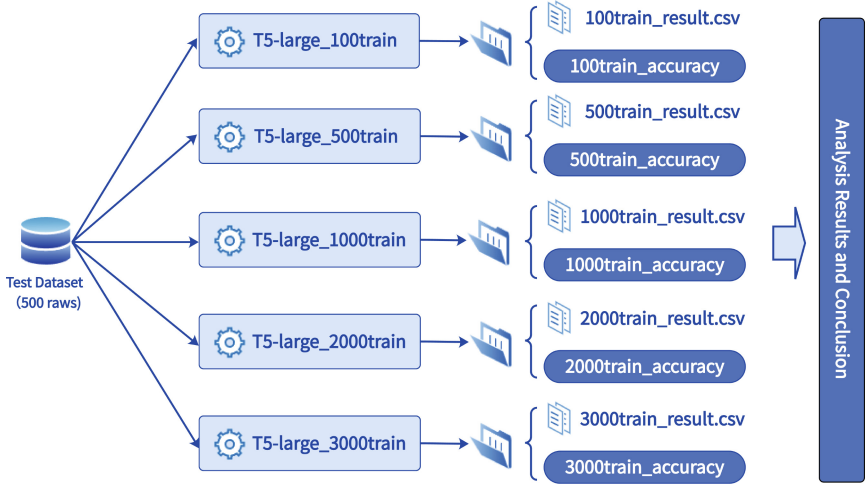


Fig. 7. Testing the Fine-tuned T5-Large Model

4.4 Evaluation

Each model after fine-tuning or a few-shot learning converts the NL in the test dataset to a CTL, and the CTL generated by the conversion is exactly the same as the CTL corresponding to the NL in the dataset then $current_{count} + 1$, and $current_{count}$ is used to record the amount of data that has been successfully converted from NL to CTL by LLMs at that time. When all the data in the test dataset has been converted, calculate the test accuracy:

$$Accuracy = count_{correct} / (total + 1) \quad (4)$$

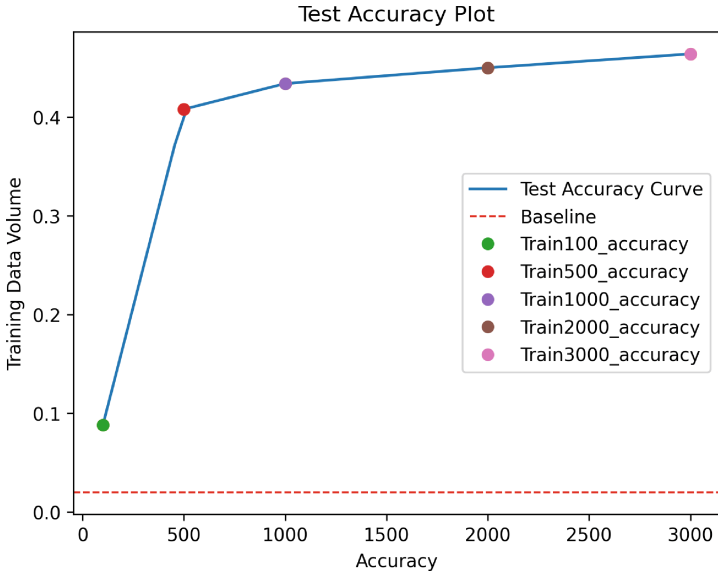
where $total$ is the total number of test data, i.e. 500. Obviously, the higher the accuracy, the better the ability of LLM to convert NL to CTL. For the stability and reliability of the results, we carried out the same test for each model three times, and took the average of all the test accuracy as the final test accuracy.

4.5 Experimental Results

As shown in Table 1, after the experiment the test accuracy of T5-Large_train3000 reaches 47% on average. T5-Large_train2000, T5-Large_train1000, T5-Large_train500 and T5-Large_train100 have decreasing test accuracy in order. Among them, the test accuracy of T5-Large_train100 is the lowest, namely 2%. As shown in Fig. 8, the size of the fine-tuned dataset is positively but non-linearly correlated with the test accuracy, and as the fine-tuned dataset gets larger, the growth of its data has less impact on the test accuracy. From the table it can be seen that the test accuracy of the lowest T5-Large_train100 is still much higher than that of the GPT-3.5 after few-shot learning. It is not difficult to see that the slightly fine-tuned categorized LLM (T5-Large) will also perform better on NL2CTL than the few-shot learning large-scale LLM (GPT-3.5).

Table 1. The test accuracy of the fine-tuned LLM (T5-Large) and the prompted LLM (GPT-3.5).

LLM Under Test	Data Volume	Test Accuracy
GPT-3.5_prompt (Baseline)	0 raw	0.02
T5-Large_train100	100 rows	0.088
T5-Large_train500	500 rows	0.408
T5-Large_train1000	1000 rows	0.434
T5-Large_train2000	2000 rows	0.450
T5-Large_train3000	3000 rows	0.464

**Fig. 8.** Test Accuracy vs. Training Dataset Size Chart

5 Conclusion

In this work, we propose a framework for automatically generating CTL based on natural language requirements, leveraging LLMs, from both data generation and model training perspectives. In this process, we constructed a dataset containing approximately 3K enhanced NL-TL pairs to fine-tune the T5 model. The fine-tuned T5-Large model achieved higher accuracy compared to the baseline prompted GPT-3.5, demonstrating the feasibility of the proposed method.

In future work, we plan to improve the method for randomly generating CTLs (e.g., by increasing the number of iterations) to enhance the quality of the dataset and thereby improve the model’s accuracy. Additionally, we also plan

to explore the generation of formal models using LLMs to further automate the model checking-based formal verification methods.

Acknowledgment. This work is supported by National Key Research and Development Program (No.2022YFB3305200), National Natural Science Foundation of China Projects (No.92370201)

References

1. Achiam, J., et al.: GPT-4 technical report. arXiv preprint [arXiv:2303.08774](https://arxiv.org/abs/2303.08774) (2023)
2. Arora, C., John, G., Mohamed, A.: Advancing requirements engineering through generative AI: assessing the role of LLMs (2023). arXiv preprint [arXiv:2310.13976](https://arxiv.org/abs/2310.13976)
3. Brunello, A., Montanari, A., Reynolds, M.: Synthesis of LTL formulas from natural language texts: state of the art and research directions. In: 26th International Symposium on Temporal Representation and Reasoning (TIME 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
4. Buzhinsky, I.: Formalization of natural language requirements into temporal logics: a survey. In: 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), vol. 1, pp. 400–406. IEEE (2019)
5. Chen, M., et al.: Evaluating large language models trained on code. arXiv preprint [arXiv:2107.03374](https://arxiv.org/abs/2107.03374) (2021)
6. Chen, Y., Gandhi, R., Zhang, Y., Fan, C.: NL2TL: transforming natural languages to temporal logics using large language models. arXiv preprint [arXiv:2305.07766](https://arxiv.org/abs/2305.07766) (2023)
7. Chou, G., Berenson, D., Ozay, N.: Learning constraints from demonstrations with grid and parametric representations. *Int. J. Robot. Res.* **40**(10–11), 1255–1283 (2021)
8. Chou, G., Ozay, N., Berenson, D.: Explaining multi-stage tasks by learning temporal logic formulas from suboptimal demonstrations. arXiv preprint [arXiv:2006.02411](https://arxiv.org/abs/2006.02411) (2020)
9. Chou, G., Ozay, N., Berenson, D.: Learning constraints from locally-optimal demonstrations under cost function uncertainty. *IEEE Robot. Autom. Lett.* **5**(2), 3682–3690 (2020)
10. Chou, G., Ozay, N., Berenson, D.: Learning temporal logic formulas from sub-optimal demonstrations: theory and experiments. *Auton. Robot.* **46**(1), 149–174 (2022)
11. Cosler, M., Hahn, C., Mendoza, D., Schmitt, F., Trippel, C.: nl2spec: interactively translating unstructured natural language to temporal logics with large language models. In: Enea, C., Lal, A. (eds.) *Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part II*, pp. 383–396. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-37703-7_18
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pp. 7–15 (1998)
13. Finucane, C., Jing, G., Kress-Gazit, H.: LTLMoP: experimenting with language, temporal logic and robot control. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 1988–1993. IEEE (2010)

14. Fuggitti, F., Chakraborti, T.: NL2LTL—a Python package for converting natural language (NL) instructions to linear temporal logic (LTL) formulas. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 37, pp. 16428–16430 (2023)
15. Gavran, I., Darulova, E., Majumdar, R.: Interactive synthesis of temporal specifications from examples and natural language. In: Proceedings of the ACM on Programming Languages, vol. 4(OOPSLA), pp. 1–26 (2020)
16. Grunske, L.: Specification patterns for probabilistic quality properties. In: Proceedings of the 30th International Conference on Software Engineering, pp. 31–40 (2008)
17. Hahn, C., Schmitt, F., Tillman, J.J., Metzger, N., Siber, J., Finkbeiner, B.: Formal specifications from natural language. arXiv preprint [arXiv:2206.01962](https://arxiv.org/abs/2206.01962) (2022)
18. He, J., Bartocci, E., Ničković, D., Isakovic, H., Grosu, R.: DeepSTL: from English requirements to signal temporal logic. In: Proceedings of the 44th International Conference on Software Engineering, pp. 610–622 (2022)
19. Howard, T.M., Tellex, S., Roy, N.: A natural language planner interface for mobile manipulators. In: 2014 IEEE International Conference on Robotics and Automation (ICRA), pp. 6652–6659. IEEE (2014)
20. Hsiung, E., et al.: Generalizing to new domains by mapping natural language to lifted LTL. In: 2022 International Conference on Robotics and Automation (ICRA), pp. 3624–3630. IEEE (2022)
21. Kolahdouz-Rahimi, S., Lano, K., Lin, C.: Requirement formalisation using natural language processing and machine learning: a systematic review. arXiv preprint [arXiv:2303.13365](https://arxiv.org/abs/2303.13365) (2023)
22. Konrad, S., Cheng, B.H.C.: Automated analysis of natural language properties for UML models. In: Bruel, J.-M. (ed.) Satellite Events at the MoDELS 2005 Conference, pp. 48–57. Springer, Berlin, Heidelberg (2006). https://doi.org/10.1007/11663430_6
23. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th International Conference on Software Engineering, pp. 372–381 (2005)
24. Liu, J.X., et al.: Lang2LTL: translating natural language commands to temporal specification with large language models. In: Workshop on Language and Robotics at CoRL 2022 (2022)
25. Ouyang, L., et al.: Training language models to follow instructions with human feedback. Adv. Neural. Inf. Process. Syst. **35**, 27730–27744 (2022)
26. Patel, R., Pavlick, R., Tellex, S.: Learning to ground language to temporal logical form. In: Conference of the North American Chapter of the Association for Computational Linguistics (NAACL) (2019)
27. Raffel, C., et al.: Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. **21**(140), 1–67 (2020)
28. Raman, V., Lignos, C., Finucane, C., Lee, K.C., Marcus, M.P., Kress-Gazit, H.: Sorry Dave, I’m Afraid I Can’t Do That: explaining unachievable robot tasks using natural language. In: Robotics: Science and Systems, vol. 2, pp. 2–1. Citeseer (2013)
29. Ratnaparkhi, A.: Maximum entropy models for natural language ambiguity resolution. University of Pennsylvania (1998)
30. Saini, R., Mussbacher, G., Guo, J.L., Kienzle, J.: Automated, interactive, and traceable domain modelling empowered by artificial intelligence. Softw. Syst. Model. **21**(3), 1015–1045 (2022)
31. Scobee, D.R., Sastry, S.S.: Maximum likelihood constraint inference for inverse reinforcement learning. arXiv preprint [arXiv:1909.05477](https://arxiv.org/abs/1909.05477) (2019)

32. Shah, A., Kamath, P., Shah, J.A., Li, S.: Bayesian inference of temporal task specifications from demonstrations. In: *Advances in Neural Information Processing Systems*, vol. 31 (2018)
33. Shah, A.J.: *Interactive Robot Training for Complex Tasks*. Ph.D. thesis, Massachusetts Institute of Technology (2021)
34. Tellex, S., et al.: Approaching the symbol grounding problem with probabilistic graphical models. *AI Mag.* **32**(4), 64–76 (2011)
35. Vazquez-Chanlatte, M., Jha, S., Tiwari, A., Ho, M.K., Seshia, S.: Learning task specifications from demonstrations. In: *Advances in Neural Information Processing Systems*, vol. 31 (2018)
36. Wang, C., Ross, C., Kuo, Y.L., Katz, B., Barbu, A.: Learning a natural-language to LTL executable semantic parser for grounded robotics. In: *Conference on Robot Learning*, pp. 1706–1718. PMLR (2021)
37. Wang, G., Trimbach, C., Lee, J.K., Ho, M.K., Littman, M.L.: Teaching a robot tasks of arbitrary complexity via human feedback. In: *Proceedings of the 2020 ACM/IEEE International Conference on Human-Robot Interaction*, pp. 649–657 (2020)



Repairing Event-B Models Through Quantifier Elimination

Tsutomu Kobayashi¹(✉)  and Fuyuki Ishikawa² 

¹ Japan Aerospace Exploration Agency, Tsukuba, Japan
kobayashi.tsutomu@jaxa.jp

² National Institute of Informatics, Tokyo, Japan
f-ishikawa@nii.ac.jp

Abstract. The process of formal modelling often involves the “verify-and-repair” exploration in which modellers find necessary constraints missing after they fail to verify properties. The bottleneck in this process is figuring out how to modify predicates of the behaviour from the limited feedback from verification tools. To tackle the difficulty, we propose a method for repairing faulty events in Event-B models by generating what we call an *invariant preservative*, a predicate such that the behaviour becomes invariant-preserving if we add it to the model. Our method automatically derives the necessary condition on the invariant preservative that has limited occurrences of free variables so that it can be added to a certain part of the model. Then, our method obtains a predicate that satisfies the condition through quantifier elimination. To apply quantifier elimination to Event-B models written in a set-theoretic language, we also provide a method for encoding models into integer-based representations. We found that our method can generate missing guard predicates for mutant models constructed from different types of models. We also compare the repaired models to the original ones and discuss the usefulness of our methods in developing models.

Keywords: Event-B · Model repair · Theorem proving · Quantifier elimination

1 Introduction

Constructing correct formal models is challenging since the modeller can often refer to only incomplete and informal information about the target system, such as documents in natural languages.

To support this phase, there are modelling methods for the *design exploration* of the target system. For instance, in the Event-B method [1], (1) the modeller declares the system’s safety property and behaviour as a set of predicates structured as invariants and events, (2) the modelling environment generates proof obligations (POs) of the *invariant preservation*, i.e. the predicates declared as invariants are really inductive invariants of events, and (3) the modeller attempts

to discharge POs with the help of automatic provers; a successful proof guarantees the invariant preservation, while a failed proof indicates that the modeller should restart from (1) to repair the model. A *correct-by-construction* formal model is created through iterations of these steps.

In practice, however, this process is difficult to carry out seamlessly because the modeller struggles to find a way to repair the model (e.g. identifying missing constraints and adding them to the model) when a proof is failed. The assistance with the repair process is currently limited; the modeller has to devise a way to modify the model based on the feedback given by verification tools, such as incomplete proof trees from an automatic prover or a counterexample from a model checker.

To repair a model, the modeller can update the behaviour of the system (specified as events) or the criteria of the correctness (specified as invariants) [15]. In this paper, we focus on repairs by updating events because the modeller will firstly try to see which behaviour is necessary to meet the criteria, before changing the criteria. We discuss updating invariants in Sect. 6.3.

Existing repair methods for formal models typically use model checkers to generate concrete execution traces of the model and construct predicates from them inductively [4, 12]. Such methods suffer from state explosion if they are applied to large-scale models or if generated predicates are partial.

To address the problem, we propose a method that generates what we call an *invariant preservative (IP)*, a predicate such that the behaviour becomes invariant-preserving (repaired) if we add it to the model. We focus on formal modelling methods based on guarded commands and instantiate the method for the Event-B formalism.

The central part of our method is the automatic derivation of the weakest IP from the PO of the invariant preservation. We firstly derive the condition the IP must satisfy: it should make the model invariant-preserving, it should be the weakest, and free occurrences of variables in it should be appropriately restricted. We then use a quantifier elimination (QE) method to obtain a predicate that satisfies the derived condition and explicitly represents the constraints on the system's behaviour. Since major QE algorithms are designed for formulas of inequalities of polynomials while the language of Event-B has set-theoretic constructs (e.g. functions as sets of ordered pairs), we also provide a method to encode Event-B models written in set-theoretic language into ones written in integers and arithmetic operators.

To evaluate our method, we injected faults into Event-B models, including large-scale ones and ones using set-theoretic constructs. The result showed that our method successfully repaired all faulty models in a timely manner.

The rest of this paper is organised as follows: Sect. 2 describes the modelling and verification in the Event-B method. In Sect. 3, we define IP and its use for repair. Sections 4 and 5 elaborate on our method for generating IPs and case studies for evaluating it, respectively. In Sects. 6 and 7, we discuss how our method can be used, can be extended, and differs from existing methods. We finally conclude this study in Sect. 8.

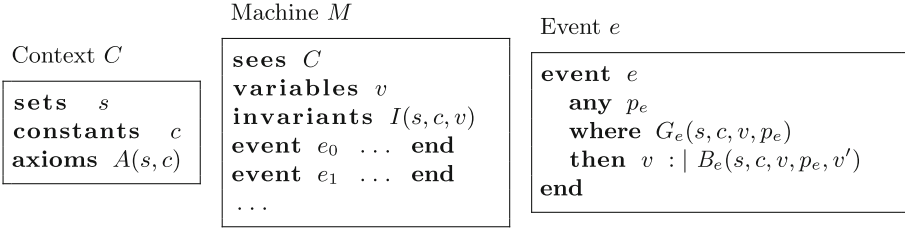


Fig. 1. Structure of Event-B model components (Color figure online)

2 Modelling and Verification in Event-B

2.1 Modelling in Event-B

Event-B [1] is a method for system-level modelling and analysis. For analysing reactive systems, the formalism is designed with influences from the Guarded Command language and Action Systems. The language of Event-B supports set-theoretic constructs, including relations and functions as sets of ordered pairs.

Components of Event-B models are shown in Fig. 1.¹ Static aspects of the target system are modelled as a *context*, which declares user-defined datatypes s , constants c , and their properties (axioms) $A(s, c)$. Dynamic aspects of the system are modelled as a *machine*, which declares the referred context C , variables v , predicates of inductive invariants $I(s, c, v)$, and state transitions as *events* $\{e_0, \dots\}$. An event is composed of parameters p_e , guard predicates $G_e(s, c, v, p_e)$, and *before-after predicates* $B_e(s, c, v, p_e, v')$. A before-after predicate describes the relation between the before-state v and the after-state v' .²

Remark 1. For a predicate to be legitimate as an invariant or a guard, free variables that occur in it must be restricted: v' (after-state variables) cannot occur free in invariants or guards because they are constraints on the current state; p_e (parameters) cannot occur free in invariants because they are global (non-event-specific) constraints.

Example 1 (Parking lot example). Consider a parking lot with a traffic light (Fig. 2). Variable n is the number of cars inside. Constant n_{capacity} is the capacity of the parking lot (@capacity_limit). Variable L is the colour of the traffic light, which can be green only when there are vacancies (invariant @grn_avail). Event `enter_unsafe` declares that a car may enter (before-after predicate @inc_n) when the traffic light is green (guard @grn). Note that variables not declared before the `:` | delimiter of **then** clause are interpreted to keep the same value as before. In this example, $L' = L$. In the next section, we show that this event does not preserve invariant @grn_avail.

¹ We omit modelling constructs not directly relevant to our method, such as ones related to refinement. See [1] for the full definition of Event-B components.

² For simplicity, we omit user-defined datatypes s and constants c from notations of predicates (e.g. A , $I(v)$) in the rest of this paper.

Context C_{parking}

```

sets color
constants green, red, ncapacity
axioms
@color: color = {green, red}
@green_red: green ≠ red
@capacity: 0 < ncapacity
    
```

```

event enter_unsafe
where
  @grn: L = green
then
  @inc_n: n :| n' = n + 1
end
    
```

 Machine M_{parking}

```

sees Cparking
variables n, L
invariants
  @types: n ∈ ℕ ∧ L ∈ color
  @capacity_limit: n ≤ ncapacity
  @grn_avail: L = green ⇒ n < ncapacity
event enter_unsafe ... end
event leave ... end
    
```

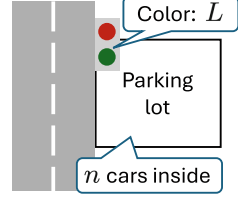


Fig. 2. Event-B model of parking lot example

2.2 Invariant Preservation

One of the primary properties of Event-B models we desire to ensure is the *invariant preservation*, i.e. the conjunction of predicates declared in the **invariants** clause is actually an inductive invariant. The preservation of the invariant by event e can be expressed as the following formula of PO:

$$\forall p_e, v, v'. A \wedge I(v) \wedge G_e(v, p_e) \wedge B_e(v, p_e, v') \implies I(v').$$

Example 2 ((Failed) preservation of invariant by enter_unsafe). The formula of the preservation of the invariant by **enter_unsafe** is as follows:

$$\begin{aligned} & \forall \text{color}, \text{green}, \text{red}, n_{\text{capacity}}, n, L, n', L'. \\ & (\quad \text{color} = \{\text{green}, \text{red}\} \wedge \text{green} \neq \text{red} \wedge 0 < n_{\text{capacity}} \\ & \quad \wedge n \in \mathbb{N} \wedge L \in \text{color} \wedge n \leq n_{\text{capacity}} \wedge (L = \text{green} \implies n < n_{\text{capacity}}) \\ & \quad \wedge L = \text{green} \wedge n' = n + 1 \wedge L' = L \\ & \implies (n' \in \mathbb{N} \wedge L' \in \text{color} \wedge n' \leq n_{\text{capacity}} \wedge (L' = \text{green} \implies n' < n_{\text{capacity}}))). \end{aligned}$$

The intuition of this PO is as follows: When the traffic light is green (this implies that the parking lot is not full), if a car enters and the light stays green, the parking lot should not become full (because the light stays green).

This formula is not valid. The counterexample is $n = n_{\text{capacity}} - 1$: in this case, incrementing n makes the parking lot full ($n' = n + 1 = n_{\text{capacity}}$), while the traffic light is kept green. Therefore, **enter_unsafe** does not preserve **@grn_avail**.

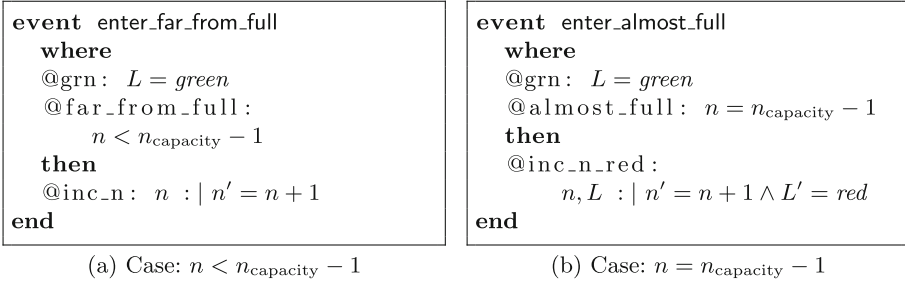


Fig. 3. Events of parking lot example (repaired)

3 Repairing Machines with Invariant Preservatives

Modelling in Event-B typically aims at exploring the design of a controller’s behaviour so that it preserves the invariant, which comes from the given requirements document. Therefore, the modeller repeats the process of constructing events, failing to prove the invariant preservation, and modifying the machine.

There are different causes of failures in discharging POs for invariant preservation [15]. The most typical cause is that some constraints are missing that properly limit the state transition not to violate the invariant. Although the invariant may sometimes turn out to be too strong or inappropriate, it is worth firstly trying to investigate constraints that make the event invariant-preserving.

In the previous example of the failed proof, we can ‘repair’ the event by adding a guard or modifying before-after predicates.

Example 3 (Repairing events by adding guards or before-after predicates). The formula of invariant preservation in Example 2 was not valid because of the possibility of the case $n = n_{capacity} - 1$. We need a case analysis here: If $n = n_{capacity} - 1$, then we should turn the traffic light red as a car enters; otherwise ($n \neq n_{capacity} - 1$), the light should be kept green. The latter is the case of $n < n_{capacity} - 1$ because of invariant $n \leq n_{capacity}$ (@capacity_limit).

To reflect this, after duplicating event `enter_unsafe` as `enter_far_from_full` and `enter_almost_full`, we can make them invariant-preserving (Fig. 3) by:

- Adding guard $n < n_{capacity} - 1$ (@far_from_full) to `enter_far_from_full`,
- Adding guard $n = n_{capacity} - 1$ (@almost_full) to `enter_almost_full`,
- Adding before-after predicate $L' = red$ (@inc_n_red) to `enter_almost_full`.

We call such predicates added to preserve invariants *invariant preservatives (IPs)*.

Definition 1 (invariant preservative). *An invariant preservative for event e is a predicate $\phi_e(v, p_e, v')$ such that:*

$$\forall p_e, v, v'. \phi_e(v, p_e, v') \wedge A \wedge I(v) \wedge G_e(v, p_e) \wedge B_e(v, p_e, v') \implies I(v').$$

We can add an IP ϕ_e for e as a guard or a before-after predicate of e :

- Adding ϕ_e as a guard of e restricts the situation where e is enabled. We call such ϕ_e *IP guard*. For instance, @far_from_full in Example 3 is an IP guard. v' should not occur in ϕ_e (Remark 1) so that ϕ_e is legitimate as a guard.
- Adding ϕ_e as a before-after predicate of e changes the state transition of e . We call such ϕ_e *IP before-after predicate*. For instance, @inc_n_red in Example 3 is an IP before-after predicate.

In practice, we want the *weakest* IP (i.e. the essential missing constraint) to maximise the debugging information. In contrast, for instance, \perp is the trivial IP that repairs every event, but it is not informative for debugging.

Although we manually devised IPs in Example 3, this process is hard, error-prone, and time-consuming in practice, especially for complex machines. Section 4 describes our method for generating the weakest IP, while adjusting free variables that occur in it, e.g. so that it can be added as a guard if we want.

Note that missing necessary predicates is not the only cause of the invariant violation. In some cases, we need to *modify* the existing event to repair it. In Sect. 6.2, we describe how our method is useful for such cases, too.

4 Methods

4.1 Overview

The overview of our method is shown in Fig. 4.

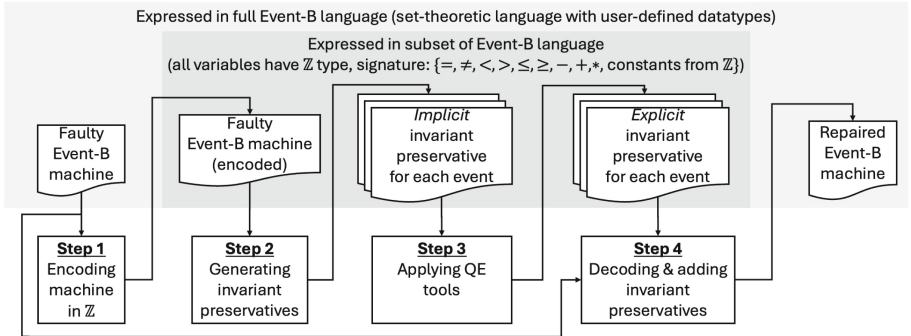


Fig. 4. Method overview

We aim at repairing a faulty Event-B machine, i.e. a machine with events that violate invariants. The method is composed of four steps:

1. We encode the target machine specified in the set-theoretic language as a machine written in integers and arithmetic operators so that QE tools will be able to handle the predicates in Step 3.

2. For each event, we generate an IP from the encoded machine. Although it satisfies the condition an IP should satisfy (Definition 1), it does not represent how the behaviour should be changed (we call it an *implicit* IP.)
3. We apply QE to obtain the predicate that satisfies the condition of implicit IP (*explicit* IP). It explicitly represents the constraint on the system's behaviour.
4. The explicit IPs are decoded to the set-theoretic language and added to the faulty machine to repair it.

The rest of this section elaborates on Steps 1–3 (Sects. 4.2–4.4) and an implementation (Sect. 4.5).

4.2 Step 1: Encoding Machines in Integers

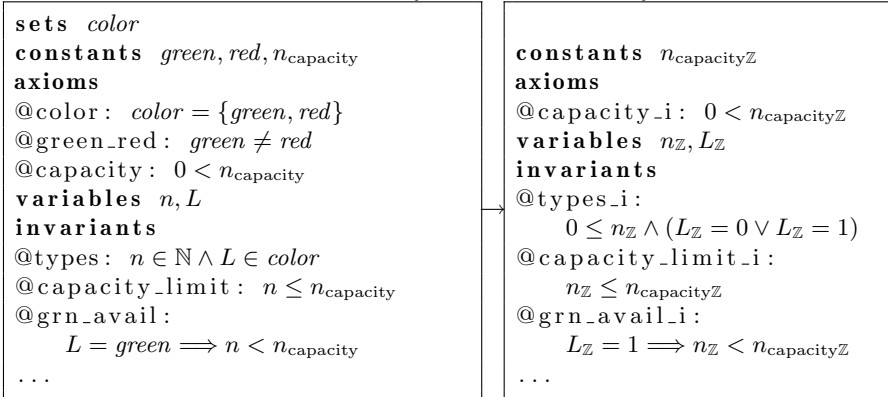
To obtain explicit IPs in Step 3, we rely on QE tools that implement QE algorithms for first-order formulas of polynomial equations and inequalities [3, 10]. For instance, Redlog [5] supports first-order formulas with the following signature: $\{=, \neq, <, >, \leq, \geq, -, +, *, (\text{constants from } \mathbb{Z})\}$.

Since Event-B machines are written in set-theoretic notations, we encode them in a subset of Event-B language so that QE tools can process the models.

Also, encoded machines tend to have simple but long sentences (e.g. Fig. 5b) to let QE tools handle them. In Sect. 6.1, we discuss possibilities of more sophisticated encoding techniques.

Encoding User-Defined Datatypes. We define a mapping from values of user-defined types to integers and use it for encoding.

Example 4 (Encoding machine M_{parking}). For encoding user-defined datatype *color*, we use the following mapping: $\{red \mapsto 0, green \mapsto 1\}$.



Encoding Pairs, Sets, Relations, and Functions. We use the following rules: (1) Ordered pairs of values are also encoded as integers using a mapping table. (2) Sets without extensional definitions are given extensional definitions with a few elements. (3) A set s is encoded as an integer i_s such that the value of i_s 's each binary digit is the value of the indicator function of s .

Example 5 (Encoding sets and functions).

We consider a simple library system for managing which books are borrowed by whom. Figure 5a shows a part of the library system written in the full Event-B language. It has user-defined datatypes *BOOKS* and *MEMBERS*, which are defined without extensional definitions; the definition simply declares there are these (possibly infinite) sets. Variable *lnd* is the lending status expressed as a partial function from *BOOKS* to *MEMBERS* (a subset of $BOOKS \times MEMBERS$). Event *lend* is about updating the lending status by adding a new pair of a book and the member who borrowed it to *lnd*.

The machine of the library system can be encoded as Fig. 5b. Firstly, *BOOKS* and *MEMBERS* are redefined as follows:

$$BOOKS = \{b_0, b_1\} \wedge b_0 \neq b_1, \quad MEMBERS = \{m_0, m_1\} \wedge m_0 \neq m_1.$$

Then, we encode *BOOKS*, *MEMBERS*, and their product as follows:

$$\begin{aligned} & \{b_0 \mapsto 0, b_1 \mapsto 1\}, \{m_0 \mapsto 0, m_1 \mapsto 1\}, \\ & \{(b_0, m_0) \mapsto 0, (b_0, m_1) \mapsto 1, (b_1, m_0) \mapsto 2, (b_1, m_1) \mapsto 3\}. \end{aligned}$$

Since *lnd* is a subset of $BOOKS \times MEMBERS = \{(b_0, m_0), (b_0, m_1), (b_1, m_0), (b_1, m_1)\}$, we encode it as a 4-bit integer (invariant @i1.i.set).

Since *lnd* is a (partial) function, (b_0, m_0) and (b_0, m_1) cannot be elements of *lnd* at the same time. Thus, in the integer encoding of *lnd* ($lnd_{\mathbb{Z}}$), the 0th bit or the 1st should be 0. The same applies to the 2nd – 3rd bits because (b_1, m_0) and (b_1, m_1) cannot be elements of *lnd* at the same time. Invariant @i1.i.pfun declares this: $lnd_{\mathbb{Z}}$ should not be ****11** or **11**** in binary notation.

Event *lend* is encoded as follows. Guard @params.type.i declares that values of parameters $book_{\mathbb{Z}}$ and $memb_{\mathbb{Z}}$ are either 0 or 1. Guard @book_not_lent.i is an encoding of @book_not_lent: $book \in \text{dom}(lnd)$. If $book_{\mathbb{Z}} = 0$ (i.e. $(book, memb)$ is either (b_0, m_0) or (b_0, m_1)), pairs (b_0, m_0) and (b_0, m_1) should not be elements of *lnd* (i.e. $lnd_{\mathbb{Z}}$ should be ****00** in binary notation.).

Before-after predicate @update_lending.i is an encoding of $lnd' = lnd \cup \{(book, memb)\}$: for instance, for $(book, memb) = (b_1, m_0)$, if the 2nd bit of $lnd_{\mathbb{Z}}$ is 0, we update it to 1 ($lnd'_{\mathbb{Z}} = lnd_{\mathbb{Z}} + 4$); otherwise, no update happens ($lnd'_{\mathbb{Z}} = lnd_{\mathbb{Z}}$).

4.3 Step 2: Generating Implicit Invariant Preservatives

In this step, for each event e , we generate an (implicit) IP for e . As we described in Sect. 3, we aim at (1) generating the weakest IP, and (2) restricting the occurrences of free variables in the IP, e.g. to let us limit the situation where e is enabled by adding a guard predicate (Remark 1).

For instance, to obtain an implicit IP guard for event e , we construct the following predicate $\gamma_e(v, p_e)$:

```

sets BOOKS, MEMBERS variables lnd
invariants @i1: lnd ∈ BOOKS → MEMBERS // →: partial function
event lend any book, memb
  where @params_type: book ∈ BOOKS ∧ memb ∈ MEMBERS
          @book_not_lent: book ∉ dom(lnd)
  then @update_lending: lnd : | lnd' = lnd ∪ {(book, memb)} end

```

(a) Before encoding

```

variables lndZ invariants @i1_i_set: 0 ≤ lndZ < 16
          @i1_i_pfun: lndZ ≠ 3 ∧ lndZ ≠ 7 ∧ lndZ ≠ 11 ∧
                    lndZ ≠ 12 ∧ lndZ ≠ 13 ∧ lndZ ≠ 14 ∧ lndZ ≠ 15
event lend any bookZ, membZ
where @params_type_i: (bookZ = 0 ∨ bookZ = 1) ∧ (membZ = 0 ∨ membZ = 1)
          @book_not_lent_i:
            (bookZ = 0 ⇒ (lndZ = 0 ∨ lndZ = 4 ∨ lndZ = 8 ∨ lndZ = 12)) ∧
            (bookZ = 1 ⇒ (lndZ = 0 ∨ lndZ = 1 ∨ lndZ = 2 ∨ lndZ = 3))
then
  @update_lending_i: lndZ : |
    (bookZ = 0 ∧ membZ = 0 ∧ (∑i∈{1,3,5,7,9,11,13,15} lndZ = i) ⇒ lnd'Z = lndZ) ∧
    (bookZ = 0 ∧ membZ = 0 ∧ (∑i∈{0,2,4,6,8,10,12,14} lndZ = i) ⇒ lnd'Z = lndZ + 1) ∧
    (bookZ = 0 ∧ membZ = 1 ∧ (∑i∈{2,3,6,7,10,11,14,15} lndZ = i) ⇒ lnd'Z = lndZ) ∧
    (bookZ = 0 ∧ membZ = 1 ∧ (∑i∈{0,1,4,5,8,9,12,13} lndZ = i) ⇒ lnd'Z = lndZ + 2) ∧
    (bookZ = 1 ∧ membZ = 0 ∧ (∑i∈{0,1,4,5,8,9,12,13,14,15} lndZ = i) ⇒ lnd'Z = lndZ) ∧
    (bookZ = 1 ∧ membZ = 0 ∧ (∑i∈{4,5,6,7,12,13,14,15} lndZ = i) ⇒ lnd'Z = lndZ + 4) ∧
    (bookZ = 1 ∧ membZ = 1 ∧ (∑i∈{0,1,2,3,8,9,10,11} lndZ = i) ⇒ lnd'Z = lndZ) ∧
    (bookZ = 1 ∧ membZ = 1 ∧ (∑i∈{8,9,10,11,12,13,14,15} lndZ = i) ⇒ lnd'Z = lndZ) ∧
    (bookZ = 1 ∧ membZ = 1 ∧ (∑i∈{0,1,2,3,4,5,6,7} lndZ = i) ⇒ lnd'Z = lndZ + 8) end

```

(b) After encoding

Fig. 5. Encoding of Library system (Example 5)

Definition 2 (Weakest invariant preservative guard). *The weakest invariant preservative guard of event e is the following predicate $\gamma_e(v, p_e)$:*

$$\gamma_e(v, p_e) := \forall v'. (A \wedge I(v) \wedge G_e(v, p_e) \wedge B_e(v, p_e, v') \implies I(v')).$$

γ_e is legitimate as a guard because primed variables (v') do not occur free in it (Remark 1).

Theorem 1. γ_e is the weakest predicate among IP guards for event e .

Proof. Let $g_e(v, p_e)$ be an arbitrary IP guard of e . Then,

$$\begin{aligned}
& \forall p_e, v, v'. (A \wedge I(v) \wedge (G_e(v, p_e) \wedge g_e(v, p_e)) \wedge B_e(v, p_e, v') \implies I(v')) \\
& \iff \forall p_e, v, v'. (g_e(v, p_e) \implies (A \wedge I(v) \wedge G_e(v, p_e) \wedge B_e(v, p_e, v') \implies I(v'))) \\
& \iff \forall p_e, v. (g_e(v, p_e) \implies \forall v'. (A \wedge I(v) \wedge G_e(v, p_e) \wedge B_e(v, p_e, v') \implies I(v'))) \\
& \iff \forall p_e, v. (g_e(v, p_e) \implies \gamma_e(v, p_e)).
\end{aligned}$$

□

Example 6 (Implicit invariant preservative guard for enter_unsafe). The weakest IP for (encoded) `enter_unsafe` from Example 1 is as follows:

$$\begin{aligned} \gamma_{\text{enter_unsafe}}(n_{\mathbb{Z}}, L_{\mathbb{Z}}, n_{\text{capacity}\mathbb{Z}}) = & \\ \forall n'_{\mathbb{Z}}, L'_{\mathbb{Z}}. (& 0 < n_{\text{capacity}\mathbb{Z}} \wedge 0 \leq n_{\mathbb{Z}} \wedge (L_{\mathbb{Z}} = 0 \vee L_{\mathbb{Z}} = 1) \wedge n_{\mathbb{Z}} \leq n_{\text{capacity}\mathbb{Z}} \\ & \wedge (L_{\mathbb{Z}} = 1 \implies n_{\mathbb{Z}} < n_{\text{capacity}\mathbb{Z}}) \wedge L_{\mathbb{Z}} = 1 \wedge n'_{\mathbb{Z}} = n_{\mathbb{Z}} + 1 \wedge L'_{\mathbb{Z}} = L_{\mathbb{Z}} \\ \implies (& 0 \leq n'_{\mathbb{Z}} \wedge (L'_{\mathbb{Z}} = 0 \vee L'_{\mathbb{Z}} = 1) \wedge n'_{\mathbb{Z}} \leq n_{\text{capacity}\mathbb{Z}} \\ & \wedge (L'_{\mathbb{Z}} = 1 \implies n'_{\mathbb{Z}} < n_{\text{capacity}\mathbb{Z}}))) . \end{aligned}$$

If we add predicate $\gamma_{\text{enter_unsafe}}$ as a guard of event `enter_unsafe`, the invariant becomes preserved by the event. However, this predicate is an implicit repair; it represents the condition on the guard to be added (“ $n'_{\mathbb{Z}}$ and $L'_{\mathbb{Z}}$ should not occur in it, and it should strengthen the constraint on the before-state so that $I(n'_{\mathbb{Z}}, L'_{\mathbb{Z}})$ will hold under given $A, I, G_{\text{enter_unsafe}}$, and $B_{\text{enter_unsafe}}$ ”) rather than *how* we should strengthen the constraint. An explicit representation of the guard that satisfies this condition is generated in the next step (Sect. 4.4).

Note that other restrictions of free variable occurrences are also possible. For example, if we bind only $n'_{\mathbb{Z}}$ (i.e. we change the quantification to $\forall n'_{\mathbb{Z}}$) to generate another implicit IP for `enter_unsafe`, the generated IP is only legitimate as a before-after predicate (because a primed variable $L'_{\mathbb{Z}}$ occurs free in it), but it describes the constraints on $n'_{\mathbb{Z}}$ without using $L'_{\mathbb{Z}}$. We discuss this in Sect. 6.2.

4.4 Step 3: Obtaining Explicit Invariant Preservatives Through Quantifier Elimination

We apply a QE tool to $\gamma_e(v, p_e)$ for generating a predicate that satisfies the condition of the implicit IP (Sect. 4.3). In other words, in this step, we obtain a predicate that explicitly represents *how* the guard should be strengthened.

Example 7 (Obtaining explicit invariant preservative guard for enter_unsafe). By applying a QE tool to the weakest IP obtained in Example 6, removing redundancy, and decoding the result, we obtain the following predicate:

$$n_{\text{capacity}} - n - 1 \neq 0.$$

Adding this to the event as its guard makes the event invariant-preserving.

This predicate is suitable as a guard because it represents the requirements for the behaviour of the system from the viewpoint of application (“incrementing n while keeping the light green should be allowed only when $n \neq n_{\text{capacity}} - 1$,”) while the predicate in Example 6 does not directly give such information.

4.5 Implementation

We implemented Steps 2–3 as a plug-in of the Rodin platform [2], which is an environment for modelling and proving in Event-B. It generates the weakest IP guard for each event of the machine selected by the user (Fig. 6).

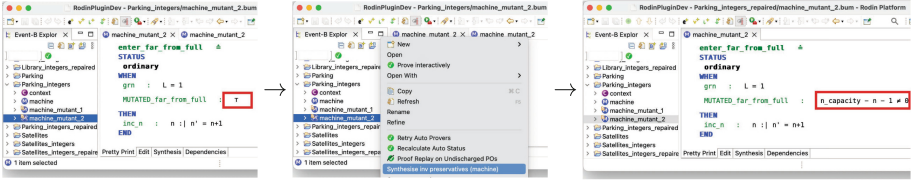


Fig. 6. Tool implementation

It uses Redlog [5] for eliminating quantifiers. It also detects and removes redundant parts of QE results, such as clauses contradictory with assumptions ($A \wedge I \wedge G$), by using the Z3 SMT solver [11].

5 Evaluation

5.1 Procedure

To evaluate our method, we injected faults to correct Event-B machines and checked whether our method could repair them by generating IP guards (Fig. 7).³

Firstly, for each target machine, we encoded the machine in the language of integers (Step 1 of the method). We carried out the encoding by manually applying systematic rules. By using the Rodin platform, we checked that the encoding was correct, i.e. all events of the encoded machines preserve invariants.

Next, we constructed *mutants* of each encoded target machine. A mutant was created by removing a guard predicate of an event. For instance, assume that we have a machine with two events shown in Fig. 3. Then our mutation procedure constructs four mutants, each of which lacks one of four guards.

Then, we used our implementation to generate the weakest IP.

Finally, in the Rodin platform, we checked that generated predicates worked as IP guards, i.e. we succeeded in proving invariant preservation after adding them to corresponding events. Moreover, using the Z3 SMT solver [11], we checked whether each generated IP was weaker than or equivalent to the removed guard under the assumptions (axioms, invariants, and existing guards).

³ The materials and detailed results are located at <https://github.com/tsutomukobayashi/ICFEM2024>.

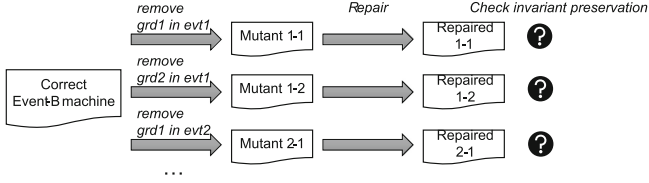


Fig. 7. Evaluation based on mutants

5.2 Materials

The target materials we used are as follows:

Cars[1, Chapter 2]. This system controls the traffic of cars on an island and the mainland, which are connected by a narrow bridge with traffic lights and sensors. The parking lot example (Example 1) is a simplified version of this.

Satellites[14]. This system models the mode logic of autonomous flight formation for satellites from the European Space Agency. We used this example to check our method’s applicability to industrial-scale models including variables of sets.

Library. A part of this system is shown in Example 5. We used this example to check our method’s applicability to models including a partial function from/to user-defined datatypes without extensional definitions.

Machine	$ V $	$ I $	$ E $	$ G $	t
Parking	2	4	3	5	< 0.6
Cars ₀	1	2	2	2	< 0.5
Cars ₁	3	5	4	6	< 1.1
Cars ₂	7	11	8	18	< 1.6
Cars ₃	18	34	16	41	< 2.5
Satellites ₀	3	7	6	13	< 0.9
Satellites ₁	5	5	10	24	< 0.8
Satellites ₂	12	55	20	78	< 9.4
Satellites ₃	20	89	32	154	< 16.2
Satellites ₄	20	7	32	134	< 7.9
Satellites ₅	22	4	32	141	< 1.0
Library	1	2	2	5	< 0.4

Table 1. Scales of materials for evaluation. $|V|$, $|I|$, $|E|$, $|G|$ respectively are numbers of variables, invariants, events, guards. t is the time (sec) taken for method Steps 2–3 per machine.

Table 1 shows the scales of the target materials (encoded faulty Event-B machines). Cars and Satellites have multiple versions of machines that correspond to different levels of abstraction. For example, Cars has four levels from the most abstract machine (Cars₀) to the most concrete one (Cars₃) that have different numbers of machine constructs.

5.3 Results

The column t of Table 1 shows the time (in seconds) spent to generate explicit IPs (Steps 2–3) per machine. Even for large-scale models such as Satellites₃, it took no more than 16.2s. Therefore, we conclude that our method is efficient enough to be used during iterative modelling. Note that the execution environment was a MacBook Pro (2021) with 64GB RAM.

After adding generated IP guards to the faulty encoded machine, we succeeded in discharging all invariant preservation POs in the Rodin platform, i.e. we succeeded in repairing all the faulty machines.

We compared the generated guards and original guards. From the semantic viewpoint, all generated guards were either equivalent to the original (encoded) ones under the assumptions (axioms, invariants, and existing guards), or weaker than the original ones. The ratio of strictly weaker guards was: 25% (Cars), 25% (Library), 50% (Parking), and 32% (Satellites).

From the syntactic viewpoint, there were several complicated guards, particularly for large-scale machines or the Library machine that use set-theoretic constructs. For instance, consider the Library machine without guard `@book_not_lent`:

```
sets BOOKS, MEMBERS variables lnd
invariants @i1: lnd ∈ BOOKS ↔ MEMBERS // ↔: partial function
event lend_mutant any book, memb
  where @params_type: book ∈ BOOKS ∧ memb ∈ MEMBERS
         // removed @book_not_lent: book ∉ dom(lnd)
  then @update_lending: lnd :| lnd' = lnd ∪ {(book, memb)} end
```

We (manually but systematically) decoded the generated guard using the mapping shown in Example 5 and obtained the following IP:

$$\begin{aligned}
 lnd &= \emptyset & (1) \\
 \vee book = b_0 \wedge (lnd = \{(b_1, m_0)\} \vee lnd = \{(b_1, m_1)\} \vee lnd = \{(b_1, m_0), (b_1, m_1)\}) & (2) \\
 \vee book = b_1 \wedge (lnd = \{(b_0, m_0)\} \vee lnd = \{(b_0, m_1)\} \vee lnd = \{(b_0, m_0), (b_0, m_1)\}) & (3) \\
 \vee book = b_0 \wedge memb = m_0 \wedge \left(\bigvee_{l \in \{(b_0, m_0)\}, \{(b_0, m_0), (b_1, m_0)\}, \{(b_0, m_0), (b_1, m_1)\}} lnd = l \right) & (4) \\
 \vee book = b_0 \wedge memb = m_1 \wedge \left(\bigvee_{l \in \{(b_0, m_1)\}, \{(b_0, m_1), (b_1, m_0)\}, \{(b_0, m_1), (b_1, m_1)\}} lnd = l \right) & (5) \\
 \vee book = b_1 \wedge memb = m_0 \wedge \left(\bigvee_{l \in \{(b_1, m_0)\}, \{(b_0, m_0), (b_1, m_0)\}, \{(b_0, m_1), (b_1, m_0)\}} lnd = l \right) & (6) \\
 \vee book = b_1 \wedge memb = m_1 \wedge \left(\bigvee_{l \in \{(b_1, m_1)\}, \{(b_0, m_0), (b_1, m_1)\}, \{(b_0, m_1), (b_1, m_1)\}} lnd = l \right). & (7)
 \end{aligned}$$

Under invariants and guards, (1)–(3) are equivalent to the original guard `@book_not_lent: book ∉ dom(lnd)`, and (4)–(7) are equivalent to $(book, memb) \in lnd$. Thus, this guard is weaker than the original one as it also allows for cases such that $(book, memb) \in lnd$, i.e. the book is already borrowed by the member. Indeed, in that case, updating lnd to $lnd \cup \{(book, memb)\}$ does not violate the invariant. In other words, the original guard is stronger as it disables the unnecessary occurrence of `lend` in which lending a book to a member who already has it because such occurrence does not change any state.

In this way, generated guards can be weaker than the original ones when the original ones reflect additional requirements represented in the form of guards. Finding such requirements missing is not within the scope of our method and generally cannot be automated. However, it is reasonably possible that modellers find necessary stronger guards by checking the generated weakest guards.

The above example also shows the fact that generated guards are sometimes complex in the encoded style. Therefore, a systematic decoding method would increase the practicality of our approach.

Findings from the evaluation: Our method successfully repaired all faulty models quickly enough to support the iterative modelling process. Generating the weakest predicates has the potential to support the case in which stronger predicates are necessary to represent missing requirements, beyond our goal of repair for invariant preservation. Although the correctness of generated predicates has been confirmed, further investigation is necessary to determine the conciseness or comprehensibility of them.

6 Discussions

6.1 Possible Extensions

Systematic Encoding/decoding Methods. Automatic or systematic encoding methods will increase the practicality of the encoding step (Sect. 4.2). Such methods can be supported by existing mathematical concepts.

For example, the Cantor pairing function $\pi(n, m) = \frac{1}{2}(n + m)(n + m + 1) + m$ can be used to encode an ordered pair of integers as an integer. Note that the pairing function is bijective, i.e. the two integers obtained by unpairing are unique.

We will investigate whether such encodings can be expressed in languages supported by QE tools and whether they can be processed with QE algorithms quickly enough.

Applicability to Other POs. In addition to the invariant preservation, Event-B has other kinds of POs. For instance, Event-B supports the construction of models with a *stepwise refinement* approach by providing special POs. Specifically, a modeller can (1) construct an abstract machine without much details of the target system, (2) construct a concrete version with more details, then (3) verify that the behaviours of two machines are in a forward simulation relation by discharging special POs. Our method can also be applied to these POs.

Extending to Other Formalisms. Although we designed our method for Event-B, the approach with slight modifications should be applicable to other variants of the Guarded Command language because the notions of guard, state transition, and verification using an inductive invariant are common.

6.2 Suggesting Various Repairs

We proposed repairing an event by adding the weakest IP guard to the event.

However, there are other repairs that may be better at clearly expressing the requirements of the target system. Therefore, various repairs need to be suggested for the practical support of model repair, so that the user can choose the best one.

In this section, we discuss how our method can be extended to generate various repairs.

Repairs as Adding IPs with Different Free Variable Occurrences. As we described in Sect. 4.3, (in addition to quantifying all primed variables,) there are other restrictions on free variable occurrences for generating IPs. It is helpful to generate multiple IPs by changing the restriction.

For instance, we can obtain multiple IPs for event `enter_unsafe` (Sect. 2.1, Example 1) as follows. Let $\psi(n, L, n', L')$ be

$$A \wedge I(n, L) \wedge G_{\text{enter_unsafe}}(n, L) \wedge B_{\text{enter_unsafe}}(n, L, n', L') \implies I(n', L').$$

Then, under the assumption of $A \wedge I \wedge G_{\text{enter_unsafe}} \wedge B_{\text{enter_unsafe}}$,

1. $\forall n', L'. \psi(n, L, n', L')$ is equivalent to $n \neq n_{\text{capacity}} - 1$.
2. $\forall n'. \psi(n, L, n', L')$ is equivalent to $n = n_{\text{capacity}} - 1 \implies L' \neq \text{green}$.

This result indicates that there are two ways to repair as we described in Sect. 3: (1) limiting the event occurrence to the case where $n \neq n_{\text{capacity}} - 1$ (as in `enter_far_from_full` (Fig. 3a)), or (2) turning the traffic light red if it is going to be full (as in `enter_almost_full` (Fig. 3b)).

Repairs as IPs with Different Strengths. Our method generates the weakest invariant preservative to provide the repair with the maximum debugging information. However, the weakest one may provide too much information.

For instance, we obtained the following guard as the IP guard for a mutant of Library example (Sect. 5.3):

$$\text{book} \notin \text{dom}(\text{lnd}) \vee (\text{book}, \text{memb}) \in \text{lnd}.$$

Although this is weaker than the original guard `@book_not_lent: book \notin dom(lnd)`, the original guard would be preferred because it meets the requirement of the library system better. Therefore, it is useful to suggest IPs that imply the weakest one. To do this, we can apply heuristics or pattern-based approaches for generating multiple candidates. For example, we can generate well-typed predicates and extract those that imply the weakest one.

Repairs as Removing Existing Predicates and Generating IPs. Although our main goal was to generate a predicate added to the event without modifying its existing predicates, we can also *modify* an event to repair it by removing predicates before generating an IP.

In some cases, we cannot obtain a meaningful IP without removing existing predicates.

Example 8 (Weakest IPs that disable event or make event infeasible). Consider the following event `evt`:

```
variables  $x$  invariants @inv:  $0 < x$ 
event evt where @grd:  $\top$  then @bap:  $x' = 0$  end
```

The weakest implicit IP guard for event `evt` is

$$\forall x'. 0 < x \wedge \top \wedge x' = 0 \implies 0 < x'.$$

By eliminating the quantifier, we obtain an additional guard $x \leq 0$. Adding this predicate as a guard *disables* the occurrence of the event because it is contradictory with invariant @inv.

The weakest IP before-after predicate for the event is

$$0 < x \wedge \top \wedge x' = 0 \implies 0 < x',$$

which is equivalent to an additional before-after predicate $x \leq 0 \vee x' \neq 0$. Adding this predicate as a before-after predicate makes the event *infeasible* because it is contradictory with invariant @inv and before-after predicate @bap.

In this case, assuming that the invariant is appropriate, this result indicates that we need to modify the existing before-after predicate @bap: $x' = 0$. If we remove @bap, the weakest IP before-after predicate for the `evt` becomes

$$0 < x \wedge \top \wedge \top \implies 0 < x'.$$

It is equivalent to $x \leq 0 \vee 0 < x'$, which can be added to `evt` as a modified version of the before-after predicate to repair the event.

As this example shows, our method can be used to repair an event by modifying it; we remove a faulty predicate of the event and generate an IP that is added to the event to replace the faulty one. The example demonstrates that our method can detect the necessity of modifying the event rather than adding missing predicates to repair it.

6.3 Applicability in Practical Contexts

We have focused on the repair approach by adding missing constraints on the behaviour to repair the model for failure in invariant preservation proofs. In practice, the cause of the failure is not limited to missing constraints. Nevertheless, our method can be applied firstly to investigate the possibilities for repair by adding missing constraints. This will cover a large part of potential causes [15] though no empirical study has yet revealed failure statistics for Event-B or formal methods.

Although our method assumes that invariants are appropriate, our method also provides insights even if the declaration of invariants is faulty. A common fault in Event-B modelling is declaring invariants that are too strong. For instance, it is common to require that $\iota(v)$ always holds, while there is an event

of exceptional behaviour; a possible repair for this case is to weaken the invariant to $\neg(mode = m_{\text{exception}}) \implies \iota(v)$. In this case, our method generates an IP that makes the event infeasible, in the same manner as Example 8, which implies that the before-after predicate of the event or the invariant should be updated.

Our future work involves developing additional repair techniques for a comprehensive repair tool. Specifically, we will pursue suggesting repairs by updating invariants. Combining such a tool with a repair guideline will also be useful.

7 Related Work

There are several studies on repairing models of Event-B (or classical B-method, whose language is similar to Event-B's). Hoang [15] classified causes of failed proofs in Event-B and provided guidelines for interpreting failed invariant preservation proofs, e.g. if the counterexample of the invariant preservation is unreachable, it means the invariant is too weak and it should be strengthened by adding an invariant predicate. However, the paper does not explain how to fix the model. The ProB model checker framework has a *disprover* [9], which efficiently generates counterexamples of the invariant preservation. Schmidt et al. [12] proposed an interactive workflow for repairing B models. The approach uses a model checker to obtain examples of traces, constructs constraints on states using pre-defined templates, and uses a constraint solver to generate a predicate for repairing the model. Cai et al. [4] proposed a tool that generates various candidates of modifications of the target model and evaluates them to pick the one with the best score. It constructs candidates by analysing concrete traces of state transitions generated by a model checker. The evaluator based on machine learning is trained to learn the state transitions of the original model, and it gives a high score to a repair candidate if its state space is close to the original's.

The primary difference between our method and those repair approaches is that they generate concrete traces of event occurrences to construct repairs, while ours does not. Because of this, our method is applicable to larger target models. Moreover, the repair constructed with our method is guaranteed to be the weakest (the most modest) among possible ones (Theorem 1).

QE has applications in a wide range of problems, including those in theorem proving and software engineering. Dolzmann et al. [6] proposed a method for automatic theorem proving of geometric problems using QE. Sturm et al. [13] proposed a method using multiple QE tools to automatically synthesise certificates for verifying hybrid dynamical systems (e.g. Lyapunov function and inductive invariant.) In Kovács and Voronkov's method [8], for a loop of a program using arrays, QE is used to extract information, which is analysed and processed by a theorem prover to obtain loop invariants. Unlike these methods, our method uses QE for repairing formal models written in set-theoretic formulas.

Automated program repair has been intensively investigated in the software engineering community [7]. Code completion or suggestion is another active area for automated support of engineers, which is being accelerated with large language models, e.g. GitHub Copilot. Those techniques for program code often

apply heuristic or learning-based “generate-and-validate” approaches. Similar automated support for engineers needs to be provided in formal methods as well while “correct-by-construction” generation is more suitable in the context of formal models as proposed in this paper.

8 Conclusion

In this paper, we tackled the problem of generating repairs of formal behavioural models, which is crucial in developing formal models through the exploration of designs. Our method generates the condition on the weakest predicate missing for preserving the invariant of a given Event-B model while controlling the occurrences of free variables in it. We then use a quantifier elimination (QE) tool to obtain a predicate that satisfies the condition. We also provide a method for systematically encoding/decoding set-theoretic predicates into an integer-based language to let QE tools handle Event-B models. For the evaluation, we constructed mutations of correct Event-B models, including complex ones, and succeeded in repairing them quickly. Our future work includes automated encoding/decoding of models and suggesting various repairs, including ones that involve modifications of the predicates existing in the target model.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Software Tools Technol. Transfer* **12**(6), 447–466 (2010). <https://doi.org/10.1007/s10009-010-0145-y>
3. Brown, C.W.: QEPCAD B: a program for computing with semi-algebraic sets using CADs. *SIGSAM Bull.* **37**(4), 97–108 (2003). <https://doi.org/10.1145/968708.968710>
4. Cai, C.H., et al.: Fast automated abstract machine repair using simultaneous modifications and refactoring. *Form. Asp. Comput.* **34**(2) (2022). <https://doi.org/10.1145/3536430>
5. Dolzmann, A., Sturm, T.: Redlog: computer algebra meets computer logic. *SIGSAM Bull.* **31**(2), 2–9 (1997). <https://doi.org/10.1145/261320.261324>
6. Dolzmann, A., Sturm, T., Weispfenning, V.: A new approach for automatic theorem proving in real geometry. *J. Autom. Reason.* **21**(3), 357–380 (1998). <https://doi.org/10.1023/A:1006031329384>
7. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: a survey. *IEEE Trans. Software Eng.* **45**(1), 34–67 (2019). <https://doi.org/10.1109/TSE.2017.2755013>
8. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00593-0_33
9. Krings, S., Bendisposto, J., Leuschel, M.: Debugging Event-B models using the ProB Disprover plug-in. In: AFADL (2007)

10. Lasaruk, A., Sturm, T.: Weak quantifier elimination for the full linear theory of the integers. *Appl. Algebra Eng. Commun. Comput.* **18**(6), 545–574 (2007). <https://doi.org/10.1007/s00200-007-0053-x>
11. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *TACAS 2008*. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
12. Schmidt, J., Krings, S., Leuschel, M.: Repair and generation of formal models using synthesis. In: Furia, C.A., Winter, K. (eds.) *IFM 2018*. LNCS, vol. 11023, pp. 346–366. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_20
13. Sturm, T., Tiwari, A.: Verification and synthesis using real quantifier elimination. In: *Proceedings of the 36th International Symposium on Symbolic and Algebraic Computation*, pp. 329–336. ISSAC '11, New York, NY, USA. Association for Computing Machinery (2011). <https://doi.org/10.1145/1993886.1993935>
14. Tarasyuk, A., Pereverzeva, I., Troubitsyna, E., Latvala, T.: The formal derivation of mode logic for autonomous satellite flight formation. In: Koornneef, F., van Gulijk, C. (eds.) *SAFECOMP 2015*. LNCS, vol. 9337, pp. 29–43. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24255-2_4
15. Thai Son Hoang: How to interpret failed proofs in Event-B. Tech. rep., ETH Zürich (2010). <https://doi.org/10.3929/ethz-a-006857374>



Tuning Trains Speed in Railway Scheduling

Étienne André^{1,2} 

¹ Université Sorbonne Paris Nord, LIPN, CNRS UMR 7030, Villetaneuse, France
ea.ndre13@lipn13.fr

² Institut Universitaire de France (IUF), Paris, France
<https://lipn.univ-paris13.fr/~andre/>

Abstract. Railway scheduling consists in ensuring that a set of trains evolve in a shared rail network without collisions, while meeting schedule constraints. This problem is notoriously difficult, even more in the case of uncertain or even unknown train speeds. We propose here a modeling and verification approach for railway scheduling in the presence of uncertain speeds, encoded here as uncertain segment durations. We formalize the system and propose a formal translation to PTAs. As a proof of concept, we apply our approach to benchmarks, for which we synthesize using IMITATOR suitable valuations for the segment durations.

Keywords: Railway scheduling · Timed automata · Parameter synthesis · IMITATOR

1 Introduction

Railway scheduling consists in ensuring that a set of trains evolve in a shared rail network without collisions, while meeting local or global, absolute or relative timing constraints. This problem is notoriously difficult, and even more in the case of uncertain or even unknown train speeds, for which the solution needs to exhibit (or *synthesize*) speeds for which the schedule constraints are met without collisions. This becomes even more tricky when the schedule constraints (specifying, e.g., the time difference between two events in the network) become themselves uncertain or unknown.

Contributions. In this paper, we offer a modeling and verification framework for railway scheduling in the presence of uncertain speeds, modeled using uncertain segment durations. Our railway model is close to that of [17] with some differences and simplifications: we consider a set of trains evolving in a shared network made of a double-vertex graph modeling segments and stations. Segments have a length and a maximum speed (which can be refined using the maximum speed of

This work is partially supported by ANR BisoUS (ANR-22-CE48-0012).

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
K. Ogata et al. (Eds.): ICFEM 2024, LNCS 15394, pp. 37–50, 2024.
https://doi.org/10.1007/978-981-96-0617-7_3

trains); such lengths and speeds are here encoded using traversal durations. Compared to [17], we notably extend the model with the ability to express uncertain or unknown speeds (and therefore durations). As target formalism for specification and verification, we choose parametric timed automata (PTAs) [4], an extension of timed automata (TAs) [3] with unknown timing constants, allowing to model variability and uncertainty. Our contributions are three-fold:

1. a formal modeling of the train trajectory problem under uncertain speeds;
2. a translation scheme from our formal model into PTAs; and
3. a set of experiments to show the applicability of our approach.

Outline. We review related works in Sect. 2. We recall necessary preliminaries in Sect. 3. We formalize our railway model (and the main problem) in Sect. 4. Our translation to PTAs is described in Sect. 5. As a proof of concept, we apply our translation to benchmarks in Sect. 6. We conclude in Sect. 7.

2 Related Works

A number of works attempt to formalize railway scheduling problems using formal methods, with different model assumptions, and different target formalisms. In [11, 22], the focus is on the formalization of railway control systems using extensions of hierarchical state machines called “Dynamic STate Machines” (DSTMs). In [24, 25], colored Petri nets are used to model railway interlocking tables, with applications to Thai railway stations. Recent works such as [17, 20] use SAT techniques, with [17] modeling continuous dynamics in a quite involved way.

Timed automata are a particularly well-suited formalism to model such problems, due to their ability to model concurrent and timed behaviors. Therefore, a number of works (such as [9, 14, 15, 18, 23, 26]) are interested in scheduling or train interlocking problems. Timing uncertainty is not considered though.

In [12], so-called parametric timed automata (differing from usual PTAs [4], as events can be parametrized too) are used to build monitors with variability in order to perform runtime verification of computer-based interlocking systems; an application to Beijing metro line 7 is briefly studied.

In contrast to these works, we address here uncertain or unknown segment traversal durations; we allow in addition for parametric schedule constraints.

Beyond the specific application to railways, planning and scheduling using TAs was considered in, e.g., [1, 2, 16]. Scheduling in the presence of uncertainty was addressed in some works using parametric timed automata, including scheduling problems with applications to the aerospace [8, 13], or schedulability under uncertainty for uniprocessor environments [5].

3 Preliminaries

We denote by \mathbb{N} , \mathbb{Z} , $\mathbb{Q}_{\geq 0}$, $\mathbb{R}_{\geq 0}$ the sets of non-negative integers, integers, non-negative rationals and non-negative reals, respectively. Let $\bowtie \in \{<, \leq, =, \geq, >\}$.

Clocks are real-valued variables that all evolve over time at the same rate. Throughout this paper, we assume a finite set $\mathbb{X} = \{x_1, \dots, x_H\}$ of *clocks*. A *clock valuation* is a function $\mu : \mathbb{X} \rightarrow \mathbb{R}_{\geq 0}$, assigning a non-negative value to each clock. We write $\mathbf{0}$ for the clock valuation assigning 0 to all clocks. Given a constant $d \in \mathbb{R}_{\geq 0}$, $\mu + d$ denotes the valuation s.t. $(\mu + d)(x) = \mu(x) + d$, for all $x \in \mathbb{X}$.

A (*timing*) *parameter* is an unknown rational-valued timing constant of a model. Throughout this paper, we assume a finite set $\mathbb{P} = \{p_1, \dots, p_M\}$ of *parameters*. A *parameter valuation* v is a function $v : \mathbb{P} \rightarrow \mathbb{Q}_{\geq 0}$.

A *parametric clock constraint* C is a conjunction of inequalities over $\mathbb{X} \cup \mathbb{P}$ of the form $x \bowtie \sum_{1 \leq i \leq |\mathbb{P}|} \alpha_i p_i + d$, with $p_i \in \mathbb{P}$, and $\alpha_i, d \in \mathbb{Z}$. Given C , we write $\mu \models v(C)$ if the expression obtained by replacing each x with $\mu(x)$ and each p with $v(p)$ in C evaluates to true.

3.1 Parametric Timed Automata

Parametric timed automata (PTAs) extend TAs with a finite set of timing parameters allowing to model unknown constants.

Definition 1 (Parametric timed automaton [4]). A PTA \mathcal{A} is a tuple $\mathcal{A} = (\Sigma, L, \ell_0, \ell_f, \mathbb{X}, \mathbb{P}, I, E)$, where:

1. Σ is a finite set of actions;
2. L is a finite set of locations;
3. $\ell_0 \in L$ is the initial location;
4. $\ell_f \in L$ is the final location;
5. \mathbb{X} is a finite set of clocks;
6. \mathbb{P} is a finite set of parameters;
7. I is the invariant, assigning to every $\ell \in L$ a parametric clock constraint $I(\ell)$ (called invariant);
8. E is a finite set of edges $e = (\ell, g, a, R, \ell')$ where $\ell, \ell' \in L$ are the source and target locations, $a \in \Sigma$, $R \subseteq \mathbb{X}$ is a set of clocks to be reset, and g is a parametric clock constraint (called guard).

As often, we assume PTAs extended with discrete global variables such as integer- or Boolean-valued variables. We also assume standard parallel composition of PTAs, synchronized on actions. The parallel composition of n PTAs is a PTA.

Definition 2 (Valuation of a PTA). Given a parameter valuation v , we denote by $v(\mathcal{A})$ the non-parametric structure where all occurrences of a parameter p_i have been replaced by $v(p_i)$.

Remark 1. We have a direct correspondence between the valuation of a PTA and the definition of a TA. TAs were originally defined with integer constants in [3], while our definition of PTAs allows *rational*-valued constants. By assuming a rescaling of the constants (i.e., by multiplying all constants in a TA by the least common multiple of their denominators), we obtain an equivalent (integer-valued) TA, as defined in [3]. So we assume in the following that $v(\mathcal{A})$ is a TA.

Definition 3 (Semantics of a TA). Given a PTA $\mathcal{A} = (\Sigma, L, \ell_0, \ell_f, \mathbb{X}, \mathbb{P}, I, E)$ and a parameter valuation v the semantics of the TA $v(\mathcal{A})$ is given by the TTS $\mathfrak{T}_{v(\mathcal{A})} = (\mathfrak{S}, \mathfrak{s}_0, \Sigma \cup \mathbb{R}_{\geq 0}, \rightarrow)$, with

1. $\mathfrak{S} = \{(\ell, \mu) \in L \times \mathbb{R}_{\geq 0}^H \mid \mu \models I(\ell)v\}$, $\mathfrak{s}_0 = (\ell_0, \mathbf{0})$,
2. \rightarrow consists of the discrete and (continuous) delay transition relations:
 - (a) discrete transitions: $(\ell, \mu) \xrightarrow{e} (\ell', \mu')$, if $(\ell, \mu), (\ell', \mu') \in \mathfrak{S}$, and there exists $e = (\ell, g, a, R, \ell') \in E$, such that $\mu' = [\mu]_R$, and $\mu \models v(g)$.
 - (b) delay transitions: $(\ell, \mu) \xrightarrow{d} (\ell, \mu + d)$, with $d \in \mathbb{R}_{\geq 0}$, if $\forall d' \in [0, d], (\ell, \mu + d') \in \mathfrak{S}$.

Moreover we write $(\ell, \mu) \xrightarrow{(d,e)} (\ell', \mu')$ for a combination of a delay and discrete transition if $\exists \mu'' : (\ell, \mu) \xrightarrow{d} (\ell, \mu'') \xrightarrow{e} (\ell', \mu')$.

Given a TA \mathcal{A} with concrete semantics $\mathfrak{T}_{\mathcal{A}}$, we refer to the states of \mathfrak{S} as the *concrete states* of \mathcal{A} . A *run* of \mathcal{A} is an alternating sequence of concrete states of \mathcal{A} and pairs of edges and delays starting from the initial state \mathfrak{s}_0 of the form $(\ell_0, \mu_0), (d_0, e_0), (\ell_1, \mu_1), \dots$ with $i = 0, 1, \dots$, $e_i \in E$, $d_i \in \mathbb{R}_{\geq 0}$ and $(\ell_i, \mu_i) \xrightarrow{(d_i, e_i)} (\ell_{i+1}, \mu_{i+1})$.

4 Railway System Model

We formalize here our railway system model. Our railway model is inspired by that of [17], with some differences that will be highlighted. A key difference is the ability of our model to define *parametric* durations. We also propose a more formal definition of the system.

4.1 Rail Network Graph

The infrastructure is modeled using a double-vertex graph [21], with nodes and segments. Nodes can be normal nodes (not allowing stopping) or stations (where trains may choose to stop or not). Segments have a length and a speed limit, encoded here using a segment traversal time (which can be exceeded for slower trains that have a speed limit smaller than the segment maximal speed). Boundary nodes are start or end nodes for the trains. As in [17], we do not model slope, angle or tunnels. However, cycles can be encoded, as opposed to [20] where this is not immediate.

We assume that segments are bidirectional, that at most one train is allowed in a segment, and that each segment is longer than any train; as a consequence, a train can occupy at most two segments at once. As in [17], “to support modeling of railway junctions, nodes of the graph have two sides (illustrated by black and blue or red colors in Fig. 1). In order to avoid physically impossible (e.g. too sharp) turns, a train has to visit both sides when transferring via such a double-sided node.” Different from [17], we model segment length and speed using a *traversal time*; similarly, since trains can occupy two segments at the same time,

we model the time needed to completely move from one segment to the next one using another traversal time. These times are minimum, as slower trains can potentially define longer times for each segment and pairs of segments (see Definition 5).

Definition 4 (rail network graph). A rail network graph is a tuple $\mathcal{G} = (N, B, St, Seg, SegDur, SegsDur, T)$ where

- N is the set of nodes,
- $B \subseteq N$ is the set of boundary nodes,
- $St \subseteq N$ is the set of stations,
- Seg is the set of segments,
- $SegDur : Seg \rightarrow \mathbb{Q}_{\geq 0} \cup \mathbb{P}$ assigns a (possibly parametric) duration to each segment,
- $SegsDur : (Seg \times Seg) \rightarrow \mathbb{Q}_{\geq 0} \cup \mathbb{P}$ assigns a (possibly parametric) duration to each pair of consecutive segments, and
- $T \in 2^{Seg} \times N \times 2^{Seg}$ is the set of transitions.

Transitions encode the way trains can move via nodes. For example, given a transition $(l, n, r) \in T$, a train can move from any segment $seg \in l$ to any segment $seg' \in r$ via n (or the opposite way).

Example 1. Consider the rail network graph in Fig. 1. The graph contains 4 boundary nodes (A, B, C, D) and 3 stations (depicted in red, and labeled with a number). Other nodes are normal nodes. Segments are labeled with a number identifying them. We can assume for example that the minimum time to traverse segment 1 is set to 8, the time to move from segment 1 to 2 is 2, while the time to move from segment 1 to 3 is 1 (values not depicted in Fig. 1).

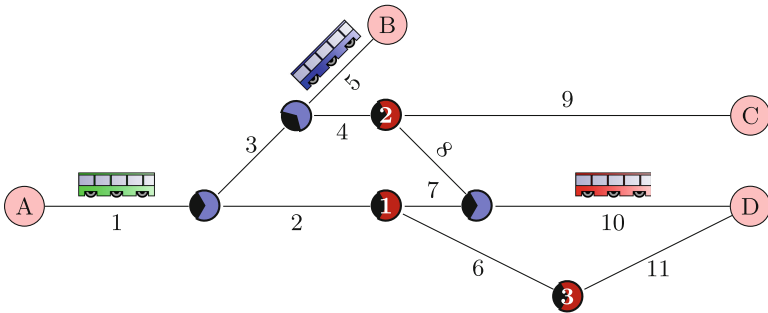


Fig. 1. An example of a rail network graph with 3 trains [17]

4.2 Trains

A train is characterized by its velocity limit and its connection. Different from [17], weight, acceleration and deceleration are not encoded; we assume they can be incorporated in segment durations. Trains always drive at their maximum possible speed; they can however stop arbitrarily long in stations.

Connection Constraints. As in [17], a *connection* is a mapping of a train to a non-empty list of nodes that must be visited in the given order. Only the first and the last element can be boundary nodes. The list of nodes must start with the boundary node denoting the starting point of the train. The list may then contain nodes that must be visited; if a node is a station, then the train can stop at this station. Trains can only stop at stations part of the connection. If the last node of the list is a boundary node, then the train must end in this node. If the last node of the connection is not a boundary node, then the train can end in any boundary node.

Each train has exactly one connection.

Example 2 (train connections [17]). Consider the green train from Fig. 1. Assume its connection is $[A, 3]$. This connection denotes that the green train must start at node A, must stop at train station 3, but cannot stop at train station 1 because it is not part of the connection. The train can end in any boundary node (even though, considering the graph topology, only D can be an end node considering the connection).

Consider the red train from Fig. 1. Assume its connection is $[D, A]$. This connection denotes that the train must depart from D, and reach A without stopping at any intermediate station; note that there are three paths allowing this connection.

Definition 5 (train). *Given a rail network graph $\mathcal{G} = (N, B, St, Seg, SegDur, SegsDur, T)$, a train over \mathcal{G} is a triple $t = (TSegDur, TSegsDur, C)$ where*

- $TSegDur : Seg \rightarrow \mathbb{Q}_{\geq 0} \cup \mathbb{P}$ assigns a possibly parametric duration to each segment,
- $TSegsDur : (Seg \times Seg) \rightarrow \mathbb{Q}_{\geq 0} \cup \mathbb{P}$ assigns a possibly parametric duration to each pair of consecutive segments, and
- $C \in N^*$ is the train connection.

Given a segment, a train drives at its maximum speed depending on the network conditions: that is, the segment duration for this train is the *maximum* between the segment duration specified by the network ($SegDur$) and the segment duration specified by the train ($TSegDur$)—and similarly for pairs of consecutive segments.

4.3 Schedule Constraints

We formalize the schedule constraints from [17], allowing to compare the time when a train arrives or departs from a node: $arrival(t, n)$ (resp. $departure(t, n)$)

denotes the time when train t arrives at (resp. leaves) node n . We use as generic notation $visit(t, n)$ to denote arrival or departure. We define three forms of constraints, detailed in the following. An originality of our approach is that we also allow for *parametric* constraints.

Ordering Constraints. Ordering constraints constrain the order in which visits should be made. They are of the form

$$visit_1(t_1, n_1) \bowtie visit_2(t_2, n_2).$$

Absolute Timing Constraints. Absolute timing constraints constrain the visit of a node at an absolute time. They are of the form

$$visit(t, n) \bowtie d, \text{ with } d \in \mathbb{Q}_{\geq 0} \cup \mathbb{P}.$$

Relative Timing Constraints. Relative timing constraints constrain the time between two visits. Let $transfer(visit_1(t_1, n_1), visit_2(t_2, n_2)) := visit_2(t_2, n_2) - visit_1(t_1, n_1)$. Then relative timing constraints are of the form

$$transfer(visit_1(t_1, n_1), visit_2(t_2, n_2)) \bowtie d, \text{ with } d \in \mathbb{Q}_{\geq 0} \cup \mathbb{P}.$$

Finally, let us define $wait(t, n) := transfer(arrival(t, n), departure(t, n))$.

Example 3 (schedule constraints). We formalize in the following some of the informal examples from [17]. The fact that the blue train must start before the green train can be encoded using $departure(t_{blue}, A) \leq departure(t_{green}, A)$. The fact that the red train starts before the green train approaches node 1 can be encoded using $departure(t_{red}, D) \leq arrival(t_{green}, 1)$. The fact that the red train must reach A within 10 time units after entering the network can be encoded using $transfer(departure(t_{red}, D), arrival(t_{red}, A)) \leq 10$. The fact that the green train must wait at node 3 for at least p time units can be encoded using $wait(t_{green}, 3) \geq p$.

4.4 Constrained Railway System

Definition 6 (constrained railway system). A constrained railway system is a tuple $\mathcal{S} = (\mathcal{G}, \mathcal{T}, \mathcal{SC})$ where

- \mathcal{G} is a rail network graph,
- \mathcal{T} is a set of trains over \mathcal{G} , and
- \mathcal{SC} is a set of schedule constraints.

4.5 Objective

Train trajectory problem under uncertain speeds:

INPUT: a constrained railway system

PROBLEM: Synthesize segment durations and schedule constraints parameters such that all train connections and schedule constraints are met.

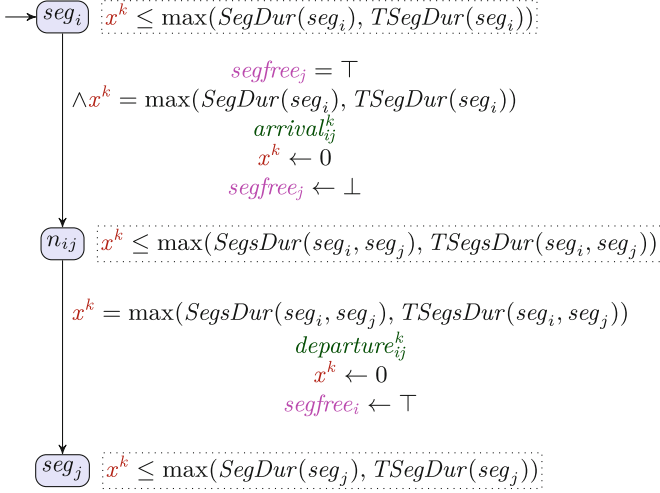


Fig. 2. Modeling two consecutive segments seg_i and seg_j via node n_{ij} for train t_k

5 Translation to Parametric Timed Automata

5.1 Overview of the Translation

Our translation is modular, in the sense that each train and each schedule constraint is translated into a different PTA. The system is made of the parallel composition of these PTAs, synchronized using the actions modeling the arrival of a train into a node, and the departure of a train from a node.

5.2 Railway Model and Trains

Due to the *concurrent* and *real-time* nature of the system, a simple discrete graph with, e.g., a list of trains currently at each node, is not a suitable approach. Instead, we choose a fully distributed approach, where each train evolves in its own representation of the rail network graph, in a continuous manner. That is, we define for each train k a PTA (with a single clock x^k), with the set of locations being made of the segments and the node of the rail network graph.

The mutual exclusion in segments and nodes is ensured using global Boolean variables, carefully tested and updated when attempting to enter, and when exiting a segment or node. More precisely, the occupancy of each segment seg_i is encoded by a Boolean variable $segfree_i$ (denoting that the segment is free).

We give in Fig. 2 the encoding of two consecutive segments seg_i and seg_j via a node n_{ij} for a given train $t_k = (\text{TSegDur}, \text{TSegsDur}, C)$. As expected, the train can remain in a segment exactly $\max(\text{SegDur}(seg_i), \text{TSegDur}(seg_i))$ time units, and similarly in a location modeling the move of a segment to the next one (here location “ n_{ij} ”). A train can move to the node between two segments only if the next segment (seg_j) is free (“ $segfree_j = \top$ ”), and the segment then

becomes occupied (“ $segfree_j \leftarrow \perp$ ”). The actions “ $arrival_{ij}^k$ ” and “ $departure_{ij}^k$ ” are used to (potentially) synchronize with PTAs modeling schedule constraints.

If the node between seg_i and seg_j is a station in which the train may stop (because it is part of its connection), then it is possible to stay longer in this node: in that case, the “=” sign in the guard between locations “ n_{ij} ” and “ seg_j ” in Fig. 2 becomes “ \geq ”, and the invariant of location “ n_{ij} ” is removed.

Connections. A connection is easily encoded using a discrete integer-valued variable: a node occurring at the n th position of the connection can only be visited if $n - 1$ nodes were visited by this train in the past, which is easily modeled by incrementing a local discrete integer. The initial location of the train PTA is the node in which the train starts, and the final location is the node the train is supposed to reach at the end of its connection.

5.3 Schedule Constraints

Ordering constraints are modeled exactly like connections, using discrete variables making sure the visits are performed in the specified order.

Each absolute timing constraint is modeled using a dedicated PTA, using a (single, global) clock x_{abs} measuring the absolute time, i.e., never reset throughout the PTAs. We give in Fig. 3a the PTA modeling constraint $visit(t, n) \bowtie d$. The PTA simply constrains action $visit_n^t$ to occur only whenever guard “ $x_{abs} \bowtie d$ ” is satisfied (recall that “ $visit$ ” stands for either “ $arrival$ ” or “ $departure$ ”).

Each relative timing constraint is modeled using a dedicated PTA, using a local clock x measuring the relative time between different events. We give in Fig. 3b the PTA modeling the relative timing constraint $transfer(visit_1(t_1, n_1), visit_2(t_2, n_2)) \bowtie d$. This PTA constrains the time difference between $visit_{n_1}^{t_1}$ and $visit_{n_2}^{t_2}$ to be as specified by the constraint, using guard “ $x \bowtie d$ ”.

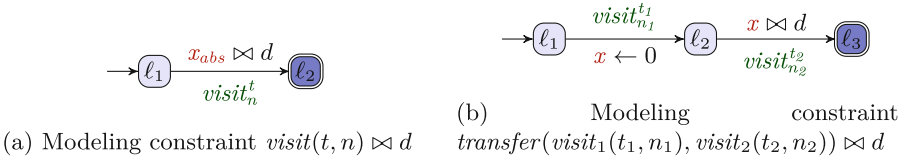


Fig. 3. Modeling schedule constraints

5.4 Solving the Train Trajectory Problem

Given \mathcal{A} the PTA resulting of the parallel composition of the aforementioned PTAs, the trajectory problem without timing parameters is satisfied if the final location of all PTAs is reachable. That is, each train reached its final destination, while all schedule constraints are satisfied.

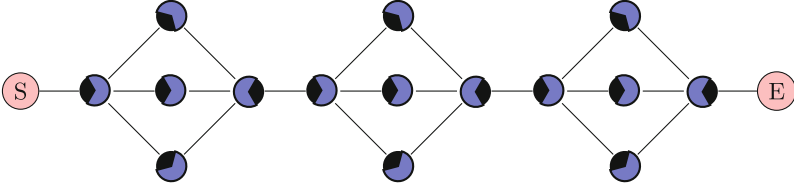


Fig. 4. An example of a serial-parallel infrastructure, with $N_S = N_P = 3$ [17]

In the presence of timing parameters, the set of parametric durations (modeling uncertain segment durations) for which all train connections and schedule constraints are met corresponds to the set of parameter valuations for which the final location of all PTAs is reachable. This can be solved using reachability-synthesis, i.e., the synthesis of timing parameters for which some PTA location is reachable [4]. While reachability-emptiness is in general undecidable [4,6], reachability-synthesis can be effectively solved under the assumption that there is no loop in the rail network graph.

6 Experiments

As a proof of concept, we verify two constrained railway systems using the IMITATOR parametric timed model checker [7]. IMITATOR takes as input networks of parametric timed automata extended with a number of features, such as synchronization and discrete variables (used here). Experiments were conducted on a Dell Precision 5570 with an Intel® Core™ i7-12700H with 32 GiB memory running Linux Mint 21 Vanessa. We used IMITATOR 3.4-alpha “Cheese Durian”, build `feat/forall_actions/788f551`. Models (including durations for Fig. 1) and results can be found at <https://www.imitator.fr/static/ICFEM24> and <https://doi.org/10.5281/zenodo.13789618>.

Running Example. We first consider the running example in Fig. 1, without timing parameters. IMITATOR derives in 1.24s that the train trajectory problem is satisfied, i.e., there exists a schedule such that all trains meet their connections and schedule constraints. Second, we add a parametric schedule constraint $visit(t_{red}, A) \leq p_R$, where $p_R \in \mathbb{P}$. That is, p_R denotes the upper bound such that the red train reaches its destination. IMITATOR derives in 1.91s the set of parameter valuations $p_R \geq 36$, i.e., the red train cannot be faster than 36 time units. Third, we parametrize the minimum duration for segments 2 and 7, i.e., $SegDur(seg_2) = p_2$ and $SegDur(seg_7) = p_7$, with $p_2, p_7 \in \mathbb{P}$. IMITATOR derives in 9.83s the set of parameter valuations $(p_R \geq p_2 + 28) \vee (p_R \geq p_2 + p_7 + 20) \vee (p_R \geq 45)$. This gives the correct (sound and complete) condition over the segment durations and schedule constraints parameters such that all train connections and schedule constraints are met. The fact that IMITATOR derives symbolic (continuous) sets of timing parameters is a major advantage over, e.g., SMT solvers, that would typically derive (non-necessarily complete)

discrete sets of specific valuations. The symbolic and dense nature of our results comes from the underlying symbolic techniques for parameter synthesis using PTAs.

Scalability. To evaluate the scalability of our approach, we consider a serial-parallel infrastructure, i.e., a network with N_S serially connected groups of N_P identical parallel tracks with a station. Connections only include $[S, E]$, i.e., trains cannot stop at any station, and are free to use any path. This variety of choices for each train obviously leads to an exponential blowup. We consider various values for the number N_T of trains in the infrastructure, and for the number of groups N_S and parallel tracks N_P ; as in [17], we fix $N_S = N_P$. An example with $N_S = N_P = 3$ is given in Fig. 4: $N_S = 3$ denotes the three groups from S to E (from left to right), while $N_P = 3$ denotes the three options (drawn vertically) to traverse one group.

We reuse two scenarios from [17]: the “nop” scenario does not contain any parametric duration, nor any schedule constraint; however, contrarily to [17], we add one parametric absolute timing constraint: $arrival(t_{N_T}, E) \leq J$, with $J \in \mathbb{P}$ (in [17], J is a constant manually tuned). That is, we measure the end-to-end time from the first train leaving S to the last train reaching E .

The “last” scenario in [17] additionally ensures that the last train takes less than bnd time units between its departure and arrival. Again, we parametrize this value instead of manually enumerating it, by adding the following relative timing constraint: $transfer(departure(t_{N_T}, S), arrival(t_{N_T}, E)) \leq bnd$, with $bnd \in \mathbb{P}$.

Note that a direct comparison with the experiments of [17] would probably not make sense since *i*) the model is not the same (on the one hand, a more involved dynamics is considered in [17] and, on the other hand, we allow for more flexible durations and schedule constraints), *ii*) the segment durations are not given in [17] and, most importantly, *iii*) we *synthesize* correct valuations while [17] only *verifies* the system for constant values.

We give in Table 1a the results for the “nop” scenario: we give from left to right the numbers of groups (and parallel tracks), of trains, of PTAs in the translated model, of clocks, of parameters, of generated states during the analysis; we finally give the synthesized value for J and the computation time. “**T.O.**” denotes timeout after 1 800 s. Similarly, we give in Table 1b the results for the “last” scenario with, as additional column, the synthesized bound “ bnd ” for the relative timing constraint.

While the computation time is clearly exponential, which is not a surprise considering the way we designed this scalability test, a positive outcome is that we get interesting results for up to 4 trains or up to 4 groups of 4 parallel tracks, a rather elaborate situation—especially in a parametric setting with unknown timing bounds in the schedule constraints.

A difference with [17, 20] is that we can automatically synthesize the bound between the first train departure and the last train arrival, while they are manually iterated in [17, 20]. The second parameter (“ bnd ”) is simply tested in [17] against 3 values ($10, 10^2, 10^3$) without attempting to synthesize a tight valuation.

Table 1. Experiments

(a) Scenario “nop”									(b) Scenario “last”								
N_S	N_T	$ \mathcal{A} $	$ \mathbb{X} $	$ \mathbb{P} $	$ \mathcal{S} $	J	$t(s)$		N_S	N_T	$ \mathcal{A} $	$ \mathbb{X} $	$ \mathbb{P} $	$ \mathcal{S} $	J	bnd	$t(s)$
2	1	2	2	1	25	63	0.01		2	1	2	3	2	25	63	63	0.01
2	2	3	3	1	238	75	0.08		2	2	3	4	2	238	75	63	0.12
2	3	4	4	1	2323	87	2.68		2	3	4	5	2	2323	87	63	3.40
2	4	5	5	1	22450	99	96.02		2	4	5	6	2	22450	99	63	113.41
3	1	2	2	1	51	92	0.01		3	1	2	3	2	51	92	92	0.02
3	2	3	3	1	1237	104	0.68		3	2	3	4	2	1052	104	92	0.91
3	3	4	4	1	27385	116	137.61		3	3	4	5	2	27385	116	92	157.78
3	4	-	-	-	-	-	T.O.		3	4	-	-	-	-	-	-	T.O.
4	1	2	2	1	87	121	0.03		4	1	2	3	2	87	121	121	0.04
4	2	3	3	1	3195	133	3.28		4	2	3	4	2	3195	133	121	4.64
4	3	-	-	-	-	-	T.O.		4	3	-	-	-	-	-	-	T.O.

7 Conclusion and Perspectives

We presented a formal model for verifying constrained railway systems in the presence of unknown durations, not only to model segment traversal times, but also to be used in relative and absolute schedule constraints. Our translation to PTAs allowed us to verify benchmarks using IMITATOR, and to derive internal segment durations and optimal values for schedule constraints.

We believe our framework, although simple, can serve as a preliminary basis for more involved settings. Notably, modeling acceleration and deceleration would be an interesting enhancement, possibly using piecewise discretization to keep the linear nature of our framework. Taking energy consumption into consideration would be another interesting future work, e.g., with an optimality criterion, perhaps integrating our setting with other approaches such as [10, 19]. In addition, tackling the exponential blowup could be partially achieved using partial order or symmetry reductions, since these models are heavily symmetric. Finally, we used here an *ad hoc* modeling language; integrating this framework into standard domain-specific languages will be an interesting extension.

Acknowledgments. I would like to thank Tomáš Kolárik and Stefan Ratschan for introducing me to their work, and an anonymous reviewer for helpful comments. The colored trains drawn using L^AT_EX TikZ in Fig. 1 are designed by [cfr](#) from [stackexchange](#).

References

1. Abdeddaïm, Y., Masson, D.: Real-time scheduling of energy harvesting embedded systems with timed automata. In: RTCSA, pp. 31–40. IEEE Computer Society (2012). <https://doi.org/10.1109/RTCSA.2012.21>

2. Abbeddaím, Y., Asarin, E., Maler, O.: Scheduling with timed automata. *TCS* **354**(2), 272–300 (2006). <https://doi.org/10.1016/j.tcs.2005.11.018>
3. Alur, R., Dill, D.L.: A theory of timed automata. *TCS* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
4. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) *STOC*, New York, NY, USA, pp. 592–601. ACM (1993). <https://doi.org/10.1145/167088.167242>
5. André, É.: A unified formalism for monoprocessor schedulability analysis under uncertainty. In: Petrucci, L., Seceleanu, C., Cavalcanti, A. (eds.) *FMICS/AVoCS-2017*. LNCS, vol. 10471, pp. 100–115. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67113-0_7
6. André, É.: What’s decidable about parametric timed automata? *Int. J. Softw. Tools Technol. Transfer* **21**(2), 203–219 (2017). <https://doi.org/10.1007/s10009-017-0467-0>
7. André, É.: IMITATOR 3: synthesis of timing parameters beyond decidability. In: Silva, A., Leino, K.R.M. (eds.) *CAV 2021*. LNCS, vol. 12759, pp. 552–565. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81685-8_26
8. André, É., Coquard, E., Fribourg, L., Jerray, J., Lesens, D.: Parametric schedulability analysis of a launcher flight control system under reactivity constraints. *FI* **182**(1), 31–67 (2021). <https://doi.org/10.3233/FI-2021-2065>
9. Avram, C., Bezerra, K., Radu, D., Machado, J., Astilean, A.: A formal approach for railroad traffic modelling using timed automata. In: Machado, J., Soares, F., Veiga, G. (eds.) *HELIX 2018*. LNEE, vol. 505, pp. 307–314. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91334-6_42
10. Bacci, G., Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Reynier, P.-A.: Optimal and robust controller synthesis using energytimed automata with uncertainty. *Formal Aspects Comput.* **33**(1), 3–25 (2020). <https://doi.org/10.1007/s00165-020-00521-4>
11. Benerecetti, M., et al.: Dynamic state machines for modelling railway control systems. *SCP* **133**, 116–153 (2017). <https://doi.org/10.1016/J.SCICO.2016.09.002>
12. Chai, M., Wang, H., Zhang, J., Tang, T.: Runtime verification of railway interlocking software with parametric timed automata. In: *ICIRT*, pp. 1–5 (2018). <https://doi.org/10.1109/ICIRT.2018.8641559>
13. Fribourg, L., Lesens, D., Moro, P., Soulat, R.: Robustness analysis for scheduling problems using the inverse method. In: Reynolds, M., Terenziani, P., Moszkowski, B. (eds.) *TIME*, pp. 73–80. IEEE Computer Society Press (2012). <https://doi.org/10.1109/TIME.2012.10>
14. Karra, S.L., Larsen, K.G., Lorber, F., Srba, J.: Safe and time-optimal control for railway games. In: Collart-Dutilleul, S., Lecomte, T., Romanovsky, A. (eds.) *RSSRail 2019*. LNCS, vol. 11495, pp. 106–122. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-18744-6_7
15. Khan, U., Ahmad, J., Saeed, T., Mirza, S.H.: On the real time modeling of interlocking system of passenger lines of Rawalpindi Cantt train station. *Complex Adapt. Syst. Model.* **4**, 17 (2016). <https://doi.org/10.1186/S40294-016-0028-5>
16. Khatib, L., Muscettola, N., Havelund, K.: Mapping temporal planning constraints into timed automata. In: *TIME*, pp. 21–27. IEEE Computer Society (2001). <https://doi.org/10.1109/TIME.2001.930693>
17. Kolárik, T., Ratschan, S.: Railway scheduling using boolean satisfiability modulo simulations. In: Chechik, M., Katoen, J.P., Leucker, M. (eds.) *FM*. LNCS, vol. 14000, pp. 56–73. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-27481-7_5

18. Laursen, P.L., Trinh, V.A.T., Haxthausen, A.E.: Formal modelling and verification of a distributed railway interlocking system using UPPAAL. In: Margaria, T., Steffen, B. (eds.) *ISoLA*, Part III. LNCS, vol. 12478, pp. 415–433. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_27
19. Lime, D., Roux, O.H., Seidner, C.: Cost problems for parametric time Petri nets. *FI* **183**(1-2), 97–123 (2021). <https://doi.org/10.3233/FI-2021-2083>
20. Luteberget, B., Claessen, K., Johansen, C., Steffen, M.: SAT modulo discrete event simulation applied to railway design capacity analysis. *FMSD* **57**(2), 211–245 (2021). <https://doi.org/10.1007/S10703-021-00368-2>
21. Montigel, M.: Formal representation of track topologies by double vertex graphs. In: *Railcomp. Computers in Railways*, vol. 2, pp. 359–370. Computational Mechanics Publications (1992)
22. Nardone, R., et al.: Dynamic state machines for formalizing railway control system specifications. In: Artho, C., Ölveczky, P.C. (eds.) *FTSCS. CCIS*, vol. 476, pp. 93–109. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-319-17581-2_7
23. Nazaruddin, Y.Y., Tamba, T.A., Pradityo, K., Aristyo, B., Widyotriatmo, A.: Safety verification of a train interlocking timed automaton model. *IFAC-PapersOnLine* **52**(15), 331–335 (2019). <https://doi.org/10.1016/j.ifacol.2019.11.696>, 8th IFAC Symposium on Mechatronic Systems MECHATRONICS 2019
24. Vanit-Anunchai, S.: Modelling railway interlocking tables using coloured Petri nets. In: Clarke, D., Agha, G.A. (eds.) *COORDINATION*. LNCS, vol. 6116, pp. 137–151. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13414-2_10
25. Vanit-Anunchai, S.: Modelling and simulating a Thai railway signalling system using coloured Petri nets. *STTT* **20**(3), 243–262 (2018). <https://doi.org/10.1007/S10009-018-0482-9>
26. Yuan, L., Wang, J., Kang, R.: The verification of temporary speed restriction of train control system based on timed automata. In: *ICCNCE*. pp. 355–358. Atlantis Press (2013). <https://doi.org/10.2991/iccnce.2013.89>



The Bright Side of Timed Opacity

Étienne André^{1,2}(✉) , Sarah Dépernet^{3,4} , and Engel Lefaucheur⁴ 

¹ Université Sorbonne Paris Nord, CNRS, Laboratoire d'Informatique de Paris Nord, LIPN, F-93430 Villetaneuse, France

eandre93430@lipn13.fr

² Institut universitaire de France (IUF), Paris, France

³ Université de Bordeaux, Bordeaux, France

⁴ Université de Lorraine, CNRS, Inria, LORIA, 54000 Nancy, France

Abstract. In 2009, Franck Cassez showed that the timed opacity problem, where an attacker can observe some actions with their timestamps and attempts to deduce information, is undecidable for timed automata (TAs). Moreover, he showed that the undecidability holds even for subclasses such as event-recording automata. In this article, we consider the same definition of opacity for several other subclasses of TAs: with restrictions on the number of clocks, of actions, on the nature of time, or on a new subclass called observable event-recording automata. We show that opacity can mostly be retrieved, except for one-action TAs and for one-clock TAs with ε -transitions, for which undecidability remains. We then exhibit a new decidable subclass in which the number of observations made by the attacker is limited.

Keywords: timed automata · opacity · timing attacks

1 Introduction

The notion of *opacity* [18, 24] concerns information leaks from a system to an attacker; that is, it expresses the power of the attacker to deduce some secret information based on some publicly observable behaviors. If an attacker observing a subset of the actions cannot deduce whether a given sequence of actions has been performed, then the system is opaque. Time particularly influences the deductive capabilities of the attacker. It has been shown in [22] that it is possible for models that are opaque when timing constraints are omitted, to become non-opaque when those constraints are added to the models.

Timed automata (TAs) [2] are an extension of finite automata that can measure and react to the passage of time, extending traditional finite automata with the ability to handle real-time constraints. They are equipped with a finite set of clocks that can be reset and compared with integer constants, enabling the modeling and verification of real-time systems.

This work is partially supported by ANR BisoUS (ANR-22-CE48-0012).

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
K. Ogata et al. (Eds.): ICFEM 2024, LNCS 15394, pp. 51–69, 2024.

https://doi.org/10.1007/978-981-96-0617-7_4

1.1 Related Work

There are several ways to define opacity problems in TAs, depending on the power of the attacker. The common idea is to ensure that the attacker cannot deduce from the observation of a run whether it was a private or a public run. The attacker in [19] is able to observe a subset $\Sigma_o \subseteq \Sigma$ of actions with their timestamps. In this context, a timed word w is said to be opaque if there exists a public run that produces the projection of w following Σ_o as an observed timed word. In this configuration, one can consider the opacity problem consisting of determining, knowing a TA \mathcal{A} and a set of timed words, whether all words in this set are opaque in \mathcal{A} . This problem has been shown to be undecidable for TAs [19]. This notably relates to the undecidability of timed language inclusion for TAs [2]. However, the undecidability holds in [19] even for the restricted class of event-recording automata (ERAs) [3] (a subclass of TAs), for which language inclusion is decidable. The aforementioned negative results leave hope only if the definition or the setting is changed, which was done in four main lines of work.

First, in [26, 27], the input model is simplified to *real-time automata* [20], a restricted formalism compared to TAs. In this setting, (initial-state) opacity becomes decidable [26, 27]. In [28], Zhang studies labeled real-timed automata (a subclass of labeled TAs); in this setting, state-based (at the initial time, the current time, etc.) opacity is proved to be decidable by extending the observer (that is, the classical powerset construction) from finite automata to labeled real-timed automata.

Second, in [5], the authors consider a time-bounded notion of the opacity of [19], where the attacker has to disclose the secret before an upper bound, using a partial observability. This can be seen as a secrecy with an *expiration date*. In addition, the analysis is carried over a time-bounded horizon. The authors prove that this problem is decidable for TAs.

Third, in [11, 12], the authors present an alternative definition to Cassez's opacity by studying *execution-time opacity*: the attacker has only access to the execution time of the system, as opposed to Cassez' partial observations with some observable events (with their timestamps). In that case, most problems become decidable (see [10] for a survey). Untimed control in this setting was considered in [7], while [11, 12] consider also *parametric* versions of the opacity problems, in which timing parameters [4] can be used in order to make the system execution-time opaque. Timed control in this setting was considered in [8].

Finally, a very recent paper (and written concurrently) [6] addresses opacity in the one-clock setting, with additional variants regarding current-location timed opacity and initial-location timed opacity. Our result regarding decidability over discrete time (Theorem 7) matches their result (see Remark 4)—we also provide exact complexity. Furthermore, our respective seemingly contradictory results on one-clock TAs without ε -transitions (we prove decidability, while undecidability is proved in [6]) are in fact not contradictory due to the presence of unobservable actions in [6] (see Remark 3).

Regarding non-interference for TAs, some decidability results are proved in [9, 15, 16], while control was considered in [17]. General security problems for TAs are surveyed in [14].

1.2 Contributions

Considering the negative results from [19] there are mainly two directions: one can consider more restrictive classes of automata, or one can limit the capabilities of the attacker—we address both directions in this work.

We address here \exists -opacity (“there exists a pair of runs, one visiting and one not visiting the private locations set, that cannot be distinguished”), weak opacity (“for any run visiting the private locations set, there is another run not visiting it and the two cannot be distinguished”) and full opacity (weak opacity, but with the other direction holding as well).

Our attacker model is as follows: the attacker knows the TA modeling the system and can observe (some) actions, but never gain access to the values of the clocks, nor knows which locations are visited. Their goal is to deduce from these observations whether a private location was visited.

Our set of contributions is threefold.

Inter-reducibility. Our first contribution is to prove that weak opacity and full opacity are inter-reducible. This result, interesting *per se*, also allows us to consider only one of both cases in the remainder of the paper.

Opacity in Subclasses of TAs. Throughout the second part of this paper (Sect. 5), we consider the same attacker settings as in [19] but for natural subclasses of TAs: first we deal with one-action TAs, then with one-clock TAs (both with and without ε -transitions—a mostly technical consideration which makes a difference in decidability), TAs over discrete time, and a new subclass which we call observable ERAs. Precisely, we show that:

1. The problem of \exists -opacity is decidable for general TAs and thus for all subclasses of TAs we consider as well (Sect. 5.1).
2. The problems of weak and full opacity are both undecidable for TAs with only one action (Sect. 5.2) or two clocks (Sect. 5.3).
3. These two problems are also undecidable for TAs with a single clock, unless we forbid ε -transitions, in which case the problems become decidable (Sect. 5.3).
4. These two problems are decidable for unrestricted TAs over discrete time (Sect. 5.4), as well as for observable ERAs (Sect. 5.5).

These results overall build on existing results from the literature. They however allow us to draw a clear border between decidability and undecidability. Moreover, we provide the exact complexity for most of the decidable results, which in some cases, complexify the proofs.

As a proof ingredient for Sect. 5.4, we also show that language inclusion is decidable for TAs over discrete time (a rather unsurprising—yet interesting—result, of which we could not find a proof in the literature).

Reducing the Attacker Power. Then, in the third part (Sect. 6), we introduce a new approach in which we reduce the visibility of the attacker to a *finite* number of actions occurring at the beginning of the run, on an unrestricted TA. This models the case of an attacker with a limited attack budget, while considering the maximal class of TAs. This more elaborate result allows us to retrieve decidability.

1.3 Outline

Section 2 recalls necessary preliminaries. Section 3 defines the problems of interest. Section 4 proves inter-reducibility of weak and full opacity. Section 5 addresses opacity for subclasses of TAs, while Sect. 6 reduces the power of the attacker to a finite set of observations. Section 7 concludes.

2 Preliminaries

We denote by $\mathbb{N}, \mathbb{Z}, \mathbb{Q}_{\geq 0}, \mathbb{R}_{\geq 0}$ the sets of non-negative integers, integers, non-negative rationals and non-negative reals, respectively. If a and b are two integers with $a \leq b$, the set $\{a, a + 1, \dots, b - 1, b\}$ is denoted by $\llbracket a; b \rrbracket$.

We let \mathbb{T} be the domain of the time, which will be either non-negative reals $\mathbb{R}_{\geq 0}$ (continuous-time semantics) or naturals \mathbb{N} (discrete-time semantics). Unless otherwise specified, we assume $\mathbb{T} = \mathbb{R}_{\geq 0}$.

Clocks are real-valued variables that all evolve over time at the same rate. Throughout this paper, we assume a set $\mathbb{X} = \{x_1, \dots, x_H\}$ of *clocks*. A *clock valuation* is a function $\mu : \mathbb{X} \rightarrow \mathbb{T}$, assigning a non-negative value to each clock. We write $\mathbf{0}$ for the clock valuation assigning 0 to all clocks. Given a constant $d \in \mathbb{T}$, $\mu + d$ denotes the valuation s.t. $(\mu + d)(x) = \mu(x) + d$, for all $x \in \mathbb{X}$. If R is a subset of \mathbb{X} and μ a clock valuation, we call *reset* of the clocks of R and denote by $[\mu]_R$ the valuation s.t. for all clock $x \in \mathbb{X}$, $[\mu]_R(x) = 0$ if $x \in R$ and $[\mu]_R(x) = \mu(x)$ otherwise.

We assume $\bowtie \in \{<, \leq, =, \geq, >\}$. A constraint C is a conjunction of inequalities over \mathbb{X} of the form $x \bowtie d$, with $d \in \mathbb{Z}$. Given C , we write $\mu \models C$ if the expression obtained by replacing each x with $\mu(x)$ in C evaluates to true.

2.1 Timed Automata

A TA is a finite automaton extended with a finite set of real-valued clocks. We also add to the standard definition of TAs a special private locations set, which is then used to define the subsequent opacity concepts.

Definition 1 (TA [2]). A TA \mathcal{A} is a tuple $\mathcal{A} = (\Sigma, L, \ell_0, L_{priv}, L_f, \mathbb{X}, I, E)$, where: 1) Σ is a finite set of actions, 2) L is a finite set of locations, $\ell_0 \in L$ is the initial location, 3) $L_{priv} \subseteq L$ is a set of private locations, $L_f \subseteq L$ is a set of final locations, 4) \mathbb{X} is a finite set of clocks, 5) I is the invariant, assigning to every $\ell \in L$ a constraint $I(\ell)$ over \mathbb{X} (called invariant), 6) E is a finite set

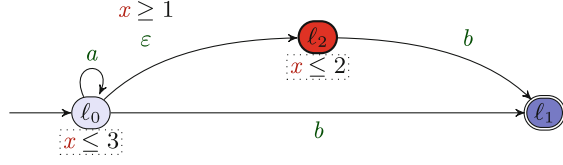


Fig. 1. A TA example

of edges $e = (\ell, g, a, R, \ell')$ where $\ell, \ell' \in L$ are the source and target locations, $a \in \Sigma \cup \{\varepsilon\}$ (where ε denotes an unobservable action), $R \subseteq \mathbb{X}$ is a set of clocks to be reset, and g is a constraint over \mathbb{X} (called guard).

Example 1. In Fig. 1, we give an example of a TA with three locations ℓ_0, ℓ_1 and ℓ_2 , three edges, two observable actions $\{a, b\}$, and one clock x . ℓ_0 is the initial location, ℓ_2 is the (unique) private location, and ℓ_1 is the (unique) final location. ℓ_0 has an invariant “ $x \leq 3$ ” and the edge from ℓ_0 to ℓ_2 is labelled by the unobservable action ε and has a guard “ $x \geq 1$ ”.

Definition 2 (Semantics of a TA). Given a TA $\mathcal{A} = (\Sigma, L, \ell_0, L_{priv}, L_f, \mathbb{X}, I, E)$, the semantics of \mathcal{A} is given by the timed transition system $\mathfrak{T}_{\mathcal{A}} = (\mathfrak{S}, \mathfrak{s}_0, \Sigma \cup \{\varepsilon\} \cup \mathbb{R}_{\geq 0}, \rightarrow)$, with

1. $\mathfrak{S} = \{(\ell, \mu) \in L \times \mathbb{R}_{\geq 0}^{\mathbb{X}} \mid \mu \models I(\ell)\}$, $\mathfrak{s}_0 = (\ell_0, \mathbf{0})$,
2. $\rightarrow \subseteq \mathfrak{S} \times E \times \mathfrak{S} \cup \mathfrak{S} \times \mathbb{R}_{\geq 0} \times \mathfrak{S}$ consists of the discrete and (continuous) delay transition relations:
 - (a) discrete transitions: $((\ell, \mu), e, (\ell', \mu')) \in \rightarrow$, and we write $(\ell, \mu) \xrightarrow{e} (\ell', \mu')$, if $(\ell, \mu), (\ell', \mu') \in \mathfrak{S}$, $e = (\ell, g, a, R, \ell') \in E$, $\mu' = [\mu]_R$, and $\mu \models g$.
 - (b) delay transitions: $((\ell, \mu), d, (\ell, \mu+d)) \in \rightarrow$, and we write $(\ell, \mu) \xrightarrow{d} (\ell, \mu+d)$, if $d \in \mathbb{R}_{\geq 0}$ and $\forall d' \in [0, d], (\ell, \mu + d') \in \mathfrak{S}$.

Moreover we write $(\ell, \mu) \xrightarrow{(d,e)} (\ell', \mu')$ for a combination of a delay and discrete transition if $\exists \mu'' : (\ell, \mu) \xrightarrow{d} (\ell, \mu'') \xrightarrow{e} (\ell', \mu')$.

Given a TA \mathcal{A} with semantics $(\mathfrak{S}, \mathfrak{s}_0, \Sigma \cup \{\varepsilon\} \cup \mathbb{R}_{\geq 0}, \rightarrow)$, we refer to the elements of \mathfrak{S} as the *configurations* of \mathcal{A} . A (finite) *run* of \mathcal{A} is an alternating sequence of configurations of \mathcal{A} and pairs of delays and edges starting from the initial configuration \mathfrak{s}_0 and ending in a final configuration (*i.e.* whose location is final), of the form $(\ell_0, \mu_0), (d_0, e_0), (\ell_1, \mu_1), \dots, (\ell_n, \mu_n)$ for some $n \in \mathbb{N}$, with $\ell_n \in L_f$ and for $i = 0, 1, \dots, n-1$, $\ell_i \notin L_f$, $e_i \in E$, $d_i \in \mathbb{R}_{\geq 0}$, and $(\ell_i, \mu_i) \xrightarrow{(d_i, e_i)} (\ell_{i+1}, \mu_{i+1})$. A *path* of \mathcal{A} is a prefix of a run ending with a configuration.

2.2 Region Automaton

We recall that the region automaton is obtained by quotienting the set of clock valuations out by an equivalence relation \simeq recalled below.

Given a TA \mathcal{A} and its set of clocks \mathbb{X} , we define $M : \mathbb{X} \rightarrow \mathbb{N}$ the map that associates to a clock x the greatest value to which the interpretations of x are compared within the guards and invariants; if x appears in no constraint, we set $M(x) = 0$.

Given $\alpha \in \mathbb{R}$, we write $\lfloor \alpha \rfloor$ and $\text{frac}(\alpha)$ respectively for the integral and fractional parts of α .

Definition 3 (Equivalence relation \simeq for valuations [2]). *Let μ, μ' be two clock valuations (with values in $\mathbb{R}_{\geq 0}$). We say that μ and μ' are equivalent, denoted by $\mu \simeq \mu'$ when, for each $x \in \mathbb{X}$, either $\mu(x) > M(x)$ and $\mu'(x) > M(x)$ or the three following conditions hold:*

1. $\lfloor \mu(x) \rfloor = \lfloor \mu'(x) \rfloor$;
2. $\text{frac}(\mu(x)) = 0$ if and only if $\text{frac}(\mu'(x)) = 0$;
3. for each $y \in \mathbb{X}$, $\text{frac}(\mu(x)) \leq \text{frac}(\mu(y))$ if and only if $\text{frac}(\mu'(x)) \leq \text{frac}(\mu'(y))$.

The equivalence relation is extended to the configurations of \mathcal{A} : let $\mathfrak{s} = (\ell, \mu)$ and $\mathfrak{s}' = (\ell', \mu')$ be two configurations in \mathcal{A} , then $\mathfrak{s} \simeq \mathfrak{s}'$ if and only if $\ell = \ell'$ and $\mu \simeq \mu'$.

The equivalence class of a valuation μ is denoted $[\mu]$ and is called a *clock region*, and the equivalence class of a configuration $\mathfrak{s} = (\ell, \mu)$ is denoted $[\mathfrak{s}]$ and called a *region* of \mathcal{A} . Clock regions are denoted by the enumeration of the constraints defining the equivalence class. Thus, values of a clock x that go beyond $M(x)$ are merged and described in the regions by “ $x > M(x)$ ”.

The set of regions of \mathcal{A} is denoted by $\mathcal{R}_{\mathcal{A}}$. These regions are of finite number: this allows us to construct a finite “untimed” regular automaton, the region automaton $\mathcal{R}\mathcal{A}_{\mathcal{A}}$. Locations of $\mathcal{R}\mathcal{A}_{\mathcal{A}}$ are regions of \mathcal{A} , and the transitions of $\mathcal{R}\mathcal{A}_{\mathcal{A}}$ convey the reachable valuations associated with each configuration in \mathcal{A} .

To formalize the construction, we need to transform discrete and time-elapsing transitions of \mathcal{A} into transitions between the regions of \mathcal{A} . To do that, we define a “time-successor” relation that corresponds to time-elapsing transitions.

Definition 4 (Time-successor relation [11]). *Let $r = (\ell, [\mu]), r' = (\ell', [\mu']) \in \mathcal{R}_{\mathcal{A}}$. We say that r' is a time-successor of r when $r \neq r'$, $\ell = \ell'$ and for each configuration (ℓ, μ) in r , there exists $d \in \mathbb{R}_{\geq 0}$ such that $(\ell, \mu + d)$ is in r' and for all $d' < d$, $(\ell, \mu + d') \in r \cup r'$.*

A region $r = (\ell, [\mu])$ is *unbounded* when, for all x in \mathbb{X} and all $\mu' \in [\mu]$, $\mu'(x) > M(x)$.

Definition 5 (Region automaton [2]). *Given a TA $\mathcal{A} = (\Sigma, L, \ell_0, L_{priv}, L_f, \mathbb{X}, I, E)$, the region automaton is the tuple $\mathcal{R}\mathcal{A}_{\mathcal{A}} = (\Sigma_{\mathcal{R}}, \mathcal{R}, r_0, \mathcal{R}_f, E_{\mathcal{R}})$ where 1) $\Sigma_{\mathcal{R}} = \Sigma \cup \{\varepsilon\}$; 2) $\mathcal{R} = \mathcal{R}_{\mathcal{A}}$; 3) $r_0 = [\mathfrak{s}_0]$; 4) \mathcal{R}_f is the set of regions whose first component is a final location $\ell_f \in L_f$; 5) i (discrete transitions) For every $r = (\ell, [\mu])$ with $\ell \notin L_f$, $r' = (\ell', [\mu']) \in \mathcal{R}_{\mathcal{A}}$ and $a \in \Sigma \cup \{\varepsilon\}$:*

$$(r, a, r') \in E_{\mathcal{R}} \text{ if } \exists \mu'' \in [\mu], \exists \mu''' \in [\mu'], (\ell, \mu'') \xrightarrow{a} (\ell', \mu''')$$

with $e = (\ell, g, a, R, \ell') \in E$.

ii) (delay transitions) For every $r = (\ell, [\mu])$ with $\ell \notin L_f$, $r' \in \mathcal{R}_A$:

$(r, \varepsilon, r') \in E_{\mathcal{R}}$ if r' is a time-successor of r or if $r = r'$ is unbounded.

As in TAs, a *run* of \mathcal{RA}_A is an alternating sequence of regions of \mathcal{RA}_A and actions starting from the initial region r_0 and ending in a final region, of the form $r_0, a_0, r_1, a_1, \dots, r_{n-1}, a_{n-1}, r_n$ for some $n \in \mathbb{N}$, with $r_n \in R_f$ and for $i \in \llbracket 0; n-1 \rrbracket$, $r_i \notin R_f$, and $(r_i, a_i, r_{i+1}) \in E_{\mathcal{R}}$. A *path* of \mathcal{RA}_A is a prefix of a run ending with a region and the trace of a path of \mathcal{RA}_A is the sequence of actions (ε excluded) contained in this path.

3 Opacity Problems in Timed Automata

3.1 Timed Words, Private and Public Runs

Given a TA \mathcal{A} and a run $\rho = (\ell_0, \mu_0), (d_0, e_0), (\ell_1, \mu_1), \dots, (\ell_n, \mu_n)$ on \mathcal{A} , we say that L_{priv} is *visited in* ρ if there exists $m \in \mathbb{N}$ such that $\ell_m \in L_{priv}$. We denote by $Visit^{priv}(\mathcal{A})$ the set of runs visiting L_{priv} , and refer to them as *private runs*. Conversely, we say that L_{priv} is *avoided in* ρ if the run ρ does not visit L_{priv} . We denote the set of runs avoiding L_{priv} by $Visit^{\overline{priv}}(\mathcal{A})$, referring to them as *public runs*.

A timed word is a sequence of pairs made up of an action and a timestamp in $\mathbb{R}_{\geq 0}$, with the timestamps being non-decreasing over the sequence. We denote by $TW^*(\Sigma)$ the set of all finite timed words over the alphabet Σ . A run ρ on a TA \mathcal{A} defines a timed word: if ρ is of the form $(\ell_0, \mu_0), (d_0, e_0), (\ell_1, \mu_1), \dots, (\ell_n, \mu_n)$ where for each $i \in \llbracket 0; n-1 \rrbracket$, $e_i = (\ell_i, g_i, a_i, R_i, \ell_{i+1})$ and $a_i \in \Sigma \cup \{\varepsilon\}$, then it generates the timed word $(a_{j_0}, \sum_{i=0}^{j_0} d_i)(a_{j_1}, \sum_{i=0}^{j_1} d_i) \cdots (a_{j_m}, \sum_{i=0}^{j_m} d_i)$, where $j_0 < j_1 < \dots < j_m$ and $\{j_k \mid k \in \llbracket 0; m \rrbracket\} = \{i \in \llbracket 0; n-1 \rrbracket \mid a_i \neq \varepsilon\}$. We denote by $Tr(\rho)$ and call *trace* of ρ the timed word generated by the run ρ and, by extension, given a set of runs Ω , we denote by $Tr(\Omega)$ the set of the traces of runs in Ω .

The set of timed words recognized by a TA \mathcal{A} is the set of traces generated by its runs, $Tr(Visit^{priv}(\mathcal{A}) \cup Visit^{\overline{priv}}(\mathcal{A}))$ (thus a subset of $(\Sigma \times \mathbb{R}_{\geq 0})^*$). To shorten these notations, we use $Tr(\mathcal{A})$ for the set of timed words recognized by \mathcal{A} , also called *language* of \mathcal{A} . Similarly, we use $Tr^{priv}(\mathcal{A}) = Tr(Visit^{priv}(\mathcal{A}))$ to denote the set of traces of private runs, and $Tr^{\overline{priv}}(\mathcal{A}) = Tr(Visit^{\overline{priv}}(\mathcal{A}))$ for the set of traces of public runs.

In Cassez's original definition [19], actions were partitioned into two sets, depending on whether an attacker could observe them or not. For simplicity, here we replaced all unobservable transition in \mathcal{A} by ε -transitions. Projecting the sequence of actions in a run onto the observable actions, as done by Cassez, is equivalent to replacing these actions by ε and taking the trace of the run. Therefore, with respect to opacity, our model is equivalent to [19].

3.2 Defining Timed Opacity

In this section, a definition of timed opacity based on the one from [19] is introduced, with three variants inspired by [10]: existential, full and weak opacity. If the attacker observes a set of runs of the system (*i.e.* observes their associated traces), we do not want them to deduce whether L_{priv} was visited or not during these observed runs. Opacity holds when the traces can be produced by both private and public runs.

We are thus first interested in the existence of an opaque trace produced by the TA, that is, a trace that cannot allow the attacker to decide whether it was generated by a private or a public run. \exists -opacity, which can be seen as the weakest form of opacity, is useful to check if there is at least one opaque trace; if not, the system cannot be made opaque by restraining the behaviors.

Definition 6 (\exists -opacity). A TA \mathcal{A} is \exists -opaque if $Tr^{priv}(\mathcal{A}) \cap Tr^{\overline{priv}}(\mathcal{A}) \neq \emptyset$.

\exists -opacity decision problem:

INPUT: A TA \mathcal{A}

PROBLEM: Is \mathcal{A} \exists -opaque?

Ideally and for a stronger security of the system, one can ask the system to be opaque *for all* possible traces of the system: a TA \mathcal{A} is fully opaque whenever for any trace in $Tr(\mathcal{A})$, it is not possible to deduce whether the run that generated this trace visited L_{priv} or not. Sometimes, a weaker notion is sufficient to ensure the required security in the system, *i.e.* when the compromising information solely comes from the identification of the private runs.

Definition 7 (Full and weak opacity). A TA \mathcal{A} is fully opaque if $Tr^{priv}(\mathcal{A}) = Tr^{\overline{priv}}(\mathcal{A})$. A TA \mathcal{A} is weakly opaque if $Tr^{priv}(\mathcal{A}) \subseteq Tr^{\overline{priv}}(\mathcal{A})$.

Full (resp. weak) opacity decision problem:

INPUT: A TA \mathcal{A}

PROBLEM: Is \mathcal{A} fully (resp. weakly) opaque?

Example 2. The TA \mathcal{A} depicted in Fig. 1 is \exists -opaque and weakly opaque but not fully opaque. Indeed,

$$Tr^{priv}(\mathcal{A}) = \{(a, \tau_1) \cdots (a, \tau_n)(b, \tau_{n+1}) \mid n \in \mathbb{N} \wedge \forall i \in \llbracket 1, n \rrbracket, \tau_i \leq \tau_{i+1} \leq 2 \wedge \tau_{n+1} \geq 1\}$$

$$Tr^{\overline{priv}}(\mathcal{A}) = \{(a, \tau_1) \cdots (a, \tau_n)(b, \tau_{n+1}) \mid n \in \mathbb{N} \wedge \forall i \in \llbracket 1, n \rrbracket, \tau_i \leq \tau_{i+1} \leq 3\}$$

This TA verifies $Tr^{priv}(\mathcal{A}) \subseteq Tr^{\overline{priv}}(\mathcal{A})$ and $Tr^{priv}(\mathcal{A}) \cap Tr^{\overline{priv}}(\mathcal{A}) \neq \emptyset$ since $(b, 1.5) \in Tr^{priv}(\mathcal{A})$.

4 Inter-reducibility of Weak and Full Opacity

In this section, we prove a new result relating weak and full opacity (Sect. 4.2). To this end, we first introduce in Sect. 4.1 a construction—that will also be useful to prove our subsequent results in Sects. 5 and 6.

4.1 \mathcal{A}_{priv} and \mathcal{A}_{pub}

First, we need a construction of two TAs \mathcal{A}_{priv} and \mathcal{A}_{pub} that recognize timed words produced respectively by private and public runs of a given TA \mathcal{A} .

The public runs TA \mathcal{A}_{pub} is the easiest to build: it suffices to remove the private locations from \mathcal{A} to eliminate every private run in the system. (See formal definition in Definition 11 in [13, Appendix A].)

The private runs TA \mathcal{A}_{priv} is obtained by duplicating all locations and transitions of \mathcal{A} : one copy \mathcal{A}_S corresponds to the paths that already visited the private locations set, and the other copy $\mathcal{A}_{\bar{S}}$ corresponds to the paths that did not (this is a usual way to encode a Boolean, here “ L_{priv} was visited”, in the locations of a TA). For each private location ℓ_{priv} in \mathcal{A} we copy all transitions leading to the copy of ℓ_{priv} in $\mathcal{A}_{\bar{S}}$ and redirect them to the copy of ℓ_{priv} in \mathcal{A}_S . The initial location is the one from $\mathcal{A}_{\bar{S}}$ and the final locations are the ones from \mathcal{A}_S . Hence all runs need to go from $\mathcal{A}_{\bar{S}}$ to \mathcal{A}_S before reaching a final location, which requires visiting a private location.

Definition 8 (Private runs TA \mathcal{A}_{priv}). Let $\mathcal{A} = (\Sigma, L, \ell_0, L_{priv}, L_f, \mathbb{X}, I, E)$ be a TA. The private runs TA $\mathcal{A}_{priv} = (\Sigma, L_S \uplus L_{\bar{S}}, \ell_0^{\bar{S}}, L_{priv}^S, L_f^S, \mathbb{X}, I', E')$ is defined as follows:

1. $L_S = \{\ell^S \mid \ell \in L\}$ and $L_{\bar{S}} = \{\ell^{\bar{S}} \mid \ell \in L\}$.
2. $L_f^S = \{\ell_f^S \mid \ell_f \in L_f\}$ is the set of final locations, and $L_{priv}^S = \{\ell_{priv}^S \mid \ell_{priv} \in L_{priv}\}$ is the set of private locations;
3. I' is defined such as $I'(\ell^S) = I'(\ell^{\bar{S}}) = I(\ell)$
4. $E' = E_S \uplus E_{\bar{S}} \uplus E_{\bar{S} \rightarrow S}$ where E_S and $E_{\bar{S}}$ are the two disjoint copies of E respectively associated with the sets of locations L_S and $L_{\bar{S}}$, and $E_{\bar{S} \rightarrow S}$ is a copy of the set of all transitions that go toward L_{priv}^S where the target location $\ell_{priv}^{\bar{S}}$ has been changed into ℓ_{priv}^S . More formally:

$$\begin{aligned} E_S &= \{(\ell^S, g, a, R, \ell'^S) \mid (\ell, g, a, R, \ell') \in E\} \\ E_{\bar{S}} &= \{(\ell^{\bar{S}}, g, a, R, \ell'^S) \mid (\ell, g, a, R, \ell') \in E\} \\ E_{\bar{S} \rightarrow S} &= \{(\ell^{\bar{S}}, g, a, R, \ell_{priv}^S) \mid (\ell, g, a, R, \ell_{priv}) \in E\}. \end{aligned}$$

Example 3. We illustrate these constructions in Fig. 2 with \mathcal{A} from Fig. 1.

The languages of \mathcal{A}_{priv} and \mathcal{A}_{pub} are respectively $Tr^{priv}(\mathcal{A})$ and $Tr^{\overline{priv}}(\mathcal{A})$.

Remark 1. By a minor modification on \mathcal{A}_{priv} , one can build a TA \mathcal{A}_{memo} that recognizes exactly the same language as \mathcal{A} and that stores in each location whether the private locations set has been visited. To do so, we add the set $\{\ell_f^{\bar{S}} \mid \ell_f \in L_f\}$ to the set of final locations in \mathcal{A}_{priv} and we remove each $\ell_{priv}^{\bar{S}} \in L_{priv}^{\bar{S}}$ from $L_{\bar{S}}$ in the same way as we did in \mathcal{A}_{pub} : the private locations of \mathcal{A}_{memo} are exactly those of \mathcal{A}_{priv} . Notably, \mathcal{A} is weakly (resp. fully) opaque if and only if \mathcal{A}_{memo} is weakly (resp. fully) opaque.

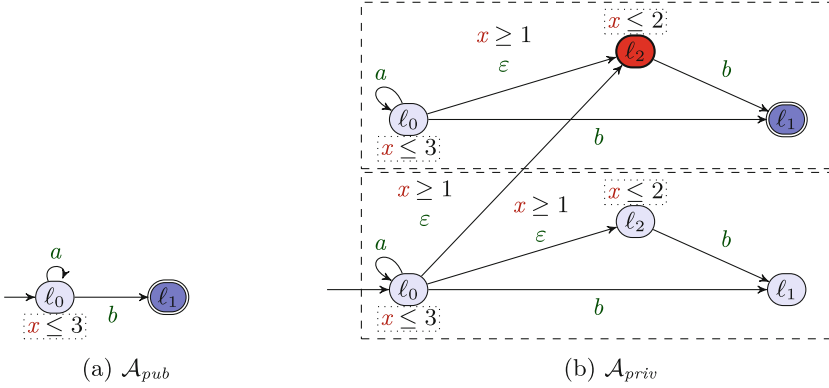


Fig. 2. \mathcal{A}_{pub} and \mathcal{A}_{priv} with the example from Fig. 1

4.2 Inter-reducibility Proof

While the distinction between weak and full notions of opacity can lead to meaningful changes [10], within our framework both associated problems are inter-reducible.

Theorem 1. *The weak opacity decision problem and the full opacity decision problem are inter-reducible.*

Proof. Let us first show that the full opacity decision problem reduces to the weak opacity decision problem. Let \mathcal{A} be a TA. In order to test whether \mathcal{A} is fully opaque, we can test both inclusions: $Tr^{priv}(\mathcal{A}) \subseteq Tr^{\overline{priv}}(\mathcal{A})$ and $Tr^{priv}(\mathcal{A}) \supseteq Tr^{\overline{priv}}(\mathcal{A})$. The first inclusion can be decided directly by testing whether \mathcal{A} is weakly opaque. In order to test the second inclusion, we need to build a TA \mathcal{B} where private and public runs are inverted. To do so, we first build \mathcal{A}_{pub} and \mathcal{A}_{priv} and then define \mathcal{B} as the TA constituted of \mathcal{A}_{pub} and \mathcal{A}_{priv} as well as two new locations ℓ'_0 and ℓ'_{priv} . The location ℓ'_0 is the initial location of \mathcal{B} and ℓ'_{priv} is the only private location. For $x \in \mathbb{X}$, both ℓ'_0 and ℓ'_{priv} have the invariant $x = 0$, ensuring no time may elapse in those locations. From ℓ'_0 , with a transition labeled by ε , one may reach either the initial location of \mathcal{A}_{priv} (ℓ_0^S) or ℓ'_{priv} , from which an ε -transition leads to the initial location of \mathcal{A}_{pub} (ℓ_0). The final locations of \mathcal{B} are the final locations of \mathcal{A}_{pub} and \mathcal{A}_{priv} . The public runs of \mathcal{B} are the ones starting in ℓ'_0 , going immediately to ℓ_0^S , and then following a run of \mathcal{A}_{priv} until a final location of \mathcal{A}_{priv} is reached. As the initial transition is labeled by ε , we have $Tr^{\overline{priv}}(\mathcal{B}) = Tr^{priv}(\mathcal{A})$. Similarly, the private runs of \mathcal{B} are the ones starting in ℓ'_0 , going immediately to ℓ'_{priv} followed immediately by going to ℓ_0^S , and then follows a run of \mathcal{A}_{pub} until a final location of \mathcal{A}_{pub} is reached. As the two initial transitions are labeled by ε , we have $Tr^{priv}(\mathcal{B}) = Tr^{\overline{priv}}(\mathcal{A})$. Hence, \mathcal{A} is fully opaque if and only if \mathcal{A} and \mathcal{B} are weakly opaque.

Let us now show the converse reduction. Let \mathcal{A} be a TA. We will define a TA \mathcal{B} such that \mathcal{B} is fully opaque if and only if \mathcal{A} is weakly opaque. To do so, we want

that $Tr^{\overline{priv}}(\mathcal{B}) = Tr^{\overline{priv}}(\mathcal{A})$ and $Tr^{priv}(\mathcal{B}) = Tr^{\overline{priv}}(\mathcal{A}) \cup Tr^{priv}(\mathcal{A})$. Indeed, if these equalities hold, $Tr^{\overline{priv}}(\mathcal{B}) = Tr^{priv}(\mathcal{B})$ would be equivalent to $Tr^{\overline{priv}}(\mathcal{A}) = Tr^{priv}(\mathcal{A}) \cup Tr^{\overline{priv}}(\mathcal{A})$ which holds if and only if $Tr^{priv}(\mathcal{A}) \subseteq Tr^{\overline{priv}}(\mathcal{A})$. As for the first reduction, \mathcal{B} contains a copy of \mathcal{A}_{pub} and \mathcal{A}_{priv} as well as two new locations ℓ'_0 and ℓ'_{priv} . The location ℓ'_0 is the initial location of \mathcal{B} and ℓ'_{priv} is the only private location. For $x \in \mathbb{X}$, both ℓ'_0 and ℓ'_{priv} have the invariant $x = 0$, ensuring no time may elapse in those locations. From ℓ'_0 , with a transition labeled by ε , one may reach either the initial location of \mathcal{A}_{pub} (ℓ_0^S) or ℓ'_{priv} , from which an ε -transition leads either to ℓ_0^S or to the initial location of \mathcal{A}_{pub} (ℓ_0). The final locations of \mathcal{B} are the final locations of \mathcal{A}_{pub} and \mathcal{A}_{priv} . The public runs of \mathcal{B} are the ones starting in ℓ'_0 , going immediately to ℓ_0 , and then following a run of \mathcal{A}_{pub} until a final location of \mathcal{A}_{pub} is reached. As the initial transition is labeled by ε , we have $Tr^{\overline{priv}}(\mathcal{B}) = Tr^{\overline{priv}}(\mathcal{A})$. Similarly, the private runs of \mathcal{B} are the ones starting in ℓ'_0 , going immediately to ℓ'_{priv} followed immediately by going to ℓ_0^S followed by a run of \mathcal{A}_{priv} , or to ℓ_0 , followed by a run of \mathcal{A}_{pub} until a final location of \mathcal{A}_{pub} is reached. As the two initial transitions are labeled by ε , we have $Tr^{priv}(\mathcal{B}) = Tr^{priv}(\mathcal{A}) \cup Tr^{\overline{priv}}(\mathcal{A})$. Hence, \mathcal{A} is weakly opaque if and only if \mathcal{B} is fully opaque. \square

5 Opacity Problems for Subclasses of Timed Automata

In this section, we consider the decidability status and complexities of the three opacity problems presented in Sect. 3 for several subclasses of TAs: TAs with one clock, TAs with one action, TAs under discrete time and observable ERAs. We first show the decidability of the \exists -opacity problem in the general case. Then, we focus on each class of TAs listed above to study weak and full opacity.

5.1 \exists -Opacity Problem

We show here (see [13, Appendix B]) that in general the \exists -opacity problem is PSPACE-complete relying on the reachability problem in TAs, which is known to be PSPACE-complete [2] as well, even for TAs with two clocks [21]. This theorem considers multiple subclasses of TAs which we will describe more in depth in future sections.

Theorem 2. *Given a TA \mathcal{A} , deciding the \exists -opacity problem for \mathcal{A} is PSPACE-complete, even when restricting \mathcal{A} to be a one-action TA, discrete-time TA, an oERA¹, or a single clock TA where integers appearing in guards are given in binary.*

If the number of clocks in \mathcal{A} is fixed and integers appearing in guards are given in unary, the \exists -opacity problem is in NLOGSPACE.

¹ See Sect. 5.5.

5.2 Timed Automata with a Single Action

Recall that the universality problem consists in deciding whether a TA \mathcal{A} accepts the set of all timed words. In [25], it is shown that the class of one-action TAs is one of the simplest cases for which the universality problem is undecidable among TAs. Therefore, this gives the intuition (see [13, Appendix C] for proof) that the weak and full opacity problems are undecidable as well for one-action TAs ($|\Sigma| = 1$).

Theorem 3. *The full and weak opacity problems for TAs with one action are undecidable.*

Remark 2. The problems of execution-time opacity introduced in [10] are a particular *decidable* subcase of these undecidable opacity problems with one-action TAs. Indeed, the execution time is equivalent to a *unique* timestamp associated with the last action of the system.

5.3 Timed Automata with a Single Clock

Following the same reasoning as in Sect. 5.2 (based on a different existing result on TAs), we show that full opacity is undecidable for one-clock TAs.

Theorem 4. *The full and weak opacity problems for one-clock TAs are undecidable.*

Proof. By reusing the same proof argument as in Theorem 3, using the fact that universality for one-clock TAs (with ε -transitions) is undecidable [1].

Without ε -Transitions. We now prove that the weak and full opacity problems become both decidable in the context of one-clock TAs ($|\mathbb{X}| = 1$) *without* ε -transitions, relying on the fact that the language inclusion problem for one-clock TAs without ε -transitions is decidable [25].

By definition, a TA is weakly opaque if $Tr^{priv}(\mathcal{A})$ is included in $Tr^{\overline{priv}}(\mathcal{A})$. As $Tr^{priv}(\mathcal{A})$ and $Tr^{\overline{priv}}(\mathcal{A})$ are respectively recognized by \mathcal{A}_{priv} and \mathcal{A}_{pub} , the decidability of the weak opacity problem is directly obtained from the decidability of the inclusion of two languages. Full opacity follows immediately, from the bidirectional language inclusion.

Theorem 5. *Full and weak opacity are decidable for one-clock TAs without ε -transitions.*

Note however that, while decidable, this problem cannot be effectively solved as the algorithm given by [25] is non-primitive recursive. Moreover, this bound is tight as shown in [1]. Hence, by imitating the approach of Theorem 3, one can reduce the language inclusion problem to the weak opacity, and thus show the complexity is tight for weak and full opacity as well.

Remark 3. This result might seem to contradict the result of a concurrently written paper [6] that proves undecidability of (language-based) opacity for one-clock TAs without ε -transitions—but it does not. The discrepancy comes from the fact that our attacker observes all actions (the unobservable actions are encoded into ε -transitions), while their setting considers unobservable actions—which can act as ε -transitions even in the absence of syntactic ε -transitions.

Now, due to the undecidability of language universality for TAs with at least two clocks [25, Theorem 21], we can prove the following with the same construction as in Theorem 3:

Theorem 6. *Full and weak opacity are undecidable for TAs with ≥ 2 clocks.*

5.4 Timed Automata over Discrete Time

In the general case, clocks are real-valued variables, with valuations thus ranging over $\mathbb{T} = \mathbb{R}_{\geq 0}$. TAs over discrete time however restrict the clock’s behavior to valuations over $\mathbb{T} = \mathbb{N}$. Since the arguments used in [2] to prove the undecidability of the universality problem in TAs rely on continuous time, this proof cannot be used to establish undecidability of opacity over discrete time. In fact, relying on the region automaton (defined in Sect. 2.2) over discrete time and classical results on finite regular automata, we show *decidability* of the opacity problems as well as their exact complexity.

If μ, μ' are two discrete clock valuations (*i.e.* with values in \mathbb{N}), the definition of \simeq from Sect. 2.2 can be simplified into: $\mu \simeq \mu'$ if and only if for each $x \in \mathbb{X}$, either $\mu(x) = \mu'(x)$ or $\mu(x) > M(x)$ and $\mu'(x) > M(x)$.

In continuous time, for each run of the TA, there is a unique corresponding run of the region automaton. In discrete time, thanks to the simplified form of the definition of \simeq , the converse statement that a run of the region automaton corresponds to a unique run of the TA nearly holds. Loss of information however remains when every clock goes beyond their maximum constant, as time elapsing is not measured beyond this point. In order to measure it, we add a letter t (for ticks) which occurs each time that an (integral) time unit passes in the region automaton. This change can be operated directly on the TA \mathcal{A} so that the correspondence between paths of \mathcal{A} and $\mathcal{RA}_{\mathcal{A}}$ becomes immediate.

More precisely, we add a clock z and add self-loop transitions $e_t = (\ell, (z = 1), t, \{z\}, \ell)$ on each location $\ell \in L$ of \mathcal{A} . We also add the guard “ $z = 0$ ” to each discrete transition of \mathcal{A} .

We illustrate the resulting TA on a simple example in Fig. 3. We depict a discrete-time TA \mathcal{A} , its transformation by the procedure we just described and finally its region automaton $\mathcal{RA}_{\mathcal{A}}$ (over discrete time).

With this construction, time information becomes superfluous in the TA as it can be deduced from the number of ticks that were produced, which also appears within a path of the region automaton. For instance, consider the run on the \mathcal{A} of Fig. 3a that remains four time units in ℓ_0 before going to ℓ_f . The timed word $(a, 4)$ on the original TA \mathcal{A} becomes $(t, 1)(t, 2)(t, 3)(t, 4)(a, 4)$ in our transformed

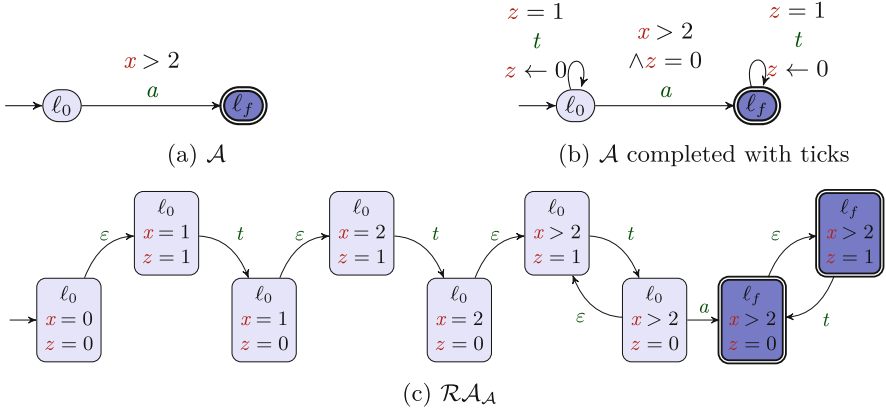


Fig. 3. A discrete-time region automaton example

TA. The untimed word obtained in $\mathcal{RA}_{\mathcal{A}}$ is $tttta$, which means that four ticks occurred before the action a was produced. From this information, the original timed word $(a, 4)$ can be reconstructed. In the rest of this subsection, we only consider TAs enhanced with ticks. From the previous discussion, we have (see [13, Appendix D]):

Lemma 1. *The language of a discrete-time TA and the language of its region automaton are in bijection.*

Thus, we show that the language inclusion problem for discrete-time TAs can be reduced to its decidable equivalent for finite regular automata.

Proposition 1. *Language inclusion in discrete-time TAs is EXPSPACE-complete.*

We can then adapt this result to the weak and full opacity problems in a similar way as done in Sect. 5.3.

Theorem 7. *Both weak and full opacity of discrete-time TAs are EXPSPACE-complete.*

Remark 4. Two very recent works [6, 23] concurrently established decidability of the opacity of TAs over discrete time. Our main distinct contribution lies in establishing the exact complexity of the problems.

5.5 Observable Event-Recording Automata

In [19], the opacity problems were shown to be undecidable for Event-Recording Automata (ERAs) [3], a subclass of TAs where every clock x is associated with a specific event a_x and x is reset on a transition if and only if this transition is labeled by a_x . Due to this, the valuations of clocks are entirely determined

by the duration since the last occurrence of the associated events. One of the main interest of ERAs is that they are determinizable [3]. This determinization is carried out through the standard subset construction.

The undecidability result from [19] on ERAs required to make the events a_x unobservable. Hence, in our framework they would be replaced by ε -transitions. We define observable ERAs (oERAs) as ERAs where the actions resetting the clocks must be observable. This means that the information required for the determinization now belongs to the trace that is observed.

Given an oERA \mathcal{A} , we can thus build through the subset construction a TA $Det_{\mathcal{A}}$ such that any path ρ in \mathcal{A} corresponds to a path ρ_D in $Det_{\mathcal{A}}$ with the same trace and ending in a location labeled by the set of all the locations of \mathcal{A} that can be reached with a run that has the same trace as ρ . This information, combined with the construction of \mathcal{A}_{memo} (Remark 1) which stores in the state of the TA whether the private location was visited or not, provides the following result (see [13, Appendix E]).

Theorem 8. *Both weak and full opacity are PSPACE-complete for oERAs.*

6 Opacity with Limited Attacker Budget

One of the causes for the undecidability of the opacity problems in [19] stems from the unbounded memory the attacker might require to remember a run of the TA. As a consequence, one can wonder whether the opacity problems remain undecidable when the attacker performs only a *finite* number of observations. This models the case of an attacker with a limited attack budget. In this section, we prove that the weak and full opacity problems become decidable whenever, given $N \in \mathbb{N}$, the attacker only observes the first N actions (with their timestamps). To the best of our knowledge, this is *i*) the second result of the literature (after [12]) providing a decidable opacity result for the full class of TAs over dense time, and *ii*) the first result limiting the number of observations of an attacker in the context of opacity for TAs.

For instance, if $(a, 1.2)(b, 1.4)(b, 1.5)(a, 2.1)$ is the trace of a public run of the system, and $N = 2$, then the attacker only observes the trace $(a, 1.2)(b, 1.4)$. If $(a, 1.2)(b, 1.4)(c, 1.6)$ is the trace of a private run, the trace observed by the attacker is $(a, 1.2)(b, 1.4)$ again and the attacker cannot conclude whether a private run occurred or not.

Formally, and in order to define new variants of opacity representing this framework, given a TA \mathcal{A} , we define a new TA (depicted in Fig. 4) which emulates the behavior of \mathcal{A} up to the N th observation. This TA is an unfolding of \mathcal{A} with $N + 1$ copies of \mathcal{A} , where ε -transitions are taken within each copy, and transitions with an observable action lead to the next copy. A run ends when either a final location or the final copy is reached.

Definition 9 (N -observation unfolding of a TA). *Let $\mathcal{A} = (\Sigma, L, \ell_0, L_{priv}, L_f, \mathbb{X}, I, E)$ be a TA and let $N \in \mathbb{N}$. We call N -unfolding of \mathcal{A} the TA $Unfold_N(\mathcal{A}) = (\Sigma, L', \ell_0^0, L'_{priv}, L'_f, \mathbb{X}, I', E')$ where*

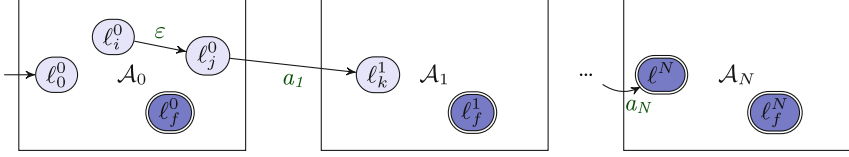


Fig. 4. The construction of an N -observation unfolded TA

1. $L' = \bigcup_{i=0}^N L^i$ where the sets L^i are $N+1$ disjoint copies of L where each location $\ell \in L$ has been renamed $\ell^i \in L^i$: for $0 \leq i \leq N$, $L^i = \{\ell^i \mid \ell \in L\}$;
2. $\ell_0^0 \in L^0$ is the initial location;
3. $L'_{priv} = \bigcup_{i=0}^{N-1} L^i_{priv}$ where L^i_{priv} are the copies within L^i of the private locations of \mathcal{A} ;
4. $L'_f = (\bigcup_{i=0}^N L^i_f) \cup L^N$ where L^i_{priv} are the copies within L^i of the final locations of \mathcal{A} ;
5. $I'(\ell^i) = I(\ell)$ for $\ell \in L$ and $i \leq N$ extends I to each L_i ;
6. $E' = \bigcup_{i=0}^{N-1} E^i \cup E^{i \rightarrow i+1}$ is the set of transitions where, given $0 \leq i < N$
 - $E^i = \{(\ell^i, \varepsilon, g, R, \ell^i) \mid (\ell, \varepsilon, g, R, \ell') \in E\}$;
 - $E^{i \rightarrow i+1} = \{(\ell^i, a, g, R, \ell^{i+1}) \mid (\ell, a, g, R, \ell') \in E \wedge a \in \Sigma\}$.

Definition 10 (Opacity w.r.t. N observations). Let \mathcal{A} be a TA and let $N \in \mathbb{N}$. We say that \mathcal{A} is weakly (resp. fully, \exists -) opaque w.r.t. N observations when $Unfold_N(\mathcal{A})$ is weakly (resp. fully, \exists -) opaque.

We now state our main result. The proof is quite technical, so we only give a high-level sketch. The full proof can be found in [13, Appendix F].

Theorem 9. The problem of deciding, given a TA \mathcal{A} and $N \in \mathbb{N}$, whether \mathcal{A} is \exists -opaque w.r.t. N observations is PSPACE-complete.

The problems of weak or full opacity w.r.t. N observations are in 2-EXPSpace.

Proof (sketch). \exists -opacity can be checked in PSPACE through the same approach as Theorem 2. Indeed, even if N is given in binary, and thus $Unfold_N(\mathcal{A})$ is of exponential size, the region automaton of $Unfold_N(\mathcal{A})$ remains simply exponential in the size of \mathcal{A} . Hardness can be achieved with $N = 0$ with the same method as Theorem 2.

Concerning the problems of weak and full opacity w.r.t. N observations, as in Sect. 5.4, our goal is to rely on the region automaton to translate the opacity problems from the TA to another problem on a finite automaton. However, there is no immediate correspondence between runs of the TA and runs of the region automaton, leading to a more involved proof.

More precisely, given a TA $\mathcal{A} = (\Sigma, L, \ell_0, L_{priv}, L_f, \mathbb{X}, I, E)$ and $N \in \mathbb{N}$, we build the unfolding of the TA \mathcal{A}_{memo} described in Remark 1. Recall that \mathcal{A}_{memo} recognizes the same language as \mathcal{A} but stores within the locations the information whether L_{priv} was visited. As such, \mathcal{A}_{memo} has the same opacity properties as \mathcal{A} , so we can consider $Unfold_N(\mathcal{A}_{memo})$ instead of $Unfold_N(\mathcal{A})$ to study the opacity of \mathcal{A} .

Additionally, we enrich this TA with ticks. In Sect. 5.4, we added a single tick to the automaton which counted the time elapsed since the start of the run. Here, the TA includes as well, for each $0 < k \leq N$, a tick clock counting the time elapsed since the k th observation. As multiple ticks may need to occur at the same time, we develop the alphabet of ticks to describe the set of tick clocks that need to be reset, *i.e.* the tick $t_{\{k_1, \dots, k_m\}}$ is produced by the TA if for every $0 \leq i \leq m$, the k_i th observation (or the start of the run if $k_i = 0$) occurred an integer number of time units before.

Note that the addition of these ticks immediately uses the assumption that only N actions are observed.

In the new ticked automaton, we will establish a correspondence between runs of the TA and paths of the region automaton, allowing us to reduce the opacity problems to non-reachability of bad states in the determinization of the region automaton, implying decidability.

Considering the complexity, the unfolding of the TA, assuming N is in binary, is exponential in the number of states. Adding the ticks means adding an exponential number of clocks as well. Hence the region automaton is doubly exponential in the original TA, and its determinization is triply exponential. Reachability being in NLOGSPACE implies the 2-EXPSPACE algorithm.

A full proof with all technical details can be found in [13, Appendix F]. \square

Table 1. Summary of Sect. 5 (\surd = decidability, \times = undecidability)

Subclass	\exists -opacity	weak opacity	full opacity
$ \Sigma = 1$		\times Theorem 3	
$ \mathbb{X} = 1$ without ε -transitions		\surd Theorem 5 (non-primitive recursive-c)	
$ \mathbb{X} = 1$	\surd Theorem 2	\times Theorem 4	
$ \mathbb{X} = 2$	(PSPACE-c)	\times Theorem 6	
$\mathbb{T} = \mathbb{N}$		\surd Theorem 7 (EXPSPACE-c)	
oERAs		\surd Theorem 8 (PSPACE-c)	

7 Conclusion and Perspectives

In this paper, we addressed three definitions of opacity on subclasses of TAs, to circumvent the undecidability from [19]. We first proved the inter-reducibility of weak and full opacity. Then, while undecidability remains for one-action TAs, we retrieve decidability for one-clock TAs without ε -transitions, or over discrete

time, or for observable ERAs. Our result for one-clock TAs without ε -transitions is tight, since we showed that increasing the number of clocks or adding ε -transitions leads to undecidability. Finally, we studied the case of an attacker with an observational power with a limited budget, *i.e.* that can only perform a finite set of observations. We proved this latter case to be decidable on the full TA formalism. We summarize the results from Sect. 5 in Table 1.

Future Work. Perspectives include being able to build a controller to ensure a TA is opaque, as well as investigating parametric versions of these problems, where timing constants considered as parameters (à la [4]) can be tuned to ensure opacity.

Finally, our result in Sect. 6 considers an attacker with a fixed attack budget; an interesting future work would be to *derive* a maximum attack budget such that the system remains opaque.

References

1. Abdulla, P.A., Deneux, J., Ouaknine, J., Quaas, K., Worrell, J.: Universality analysis for one-clock timed automata. *FI* **89**(4), 419–450 (2008)
2. Alur, R., Dill, D.L.: A theory of timed automata. *TCS* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
3. Alur, R., Fix, L., Henzinger, T.A.: Event-clock automata: a determinizable class of timed automata. *TCS* **211**(1–2), 253–273 (1999). [https://doi.org/10.1016/S0304-3975\(97\)00173-4](https://doi.org/10.1016/S0304-3975(97)00173-4)
4. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) *STOC*, pp. 592–601. ACM, New York (1993). <https://doi.org/10.1145/167088.167242>
5. Ammar, I., El Touati, Y., Yeddes, M., Mullins, J.: Bounded opacity for timed systems. *JISA* **61**, 1–13 (2021). <https://doi.org/10.1016/j.jisa.2021.102926>
6. An, J., Gao, Q., Wang, L., Zhan, N., Hasuo, I.: The opacity of timed automata. In: Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M. (eds.) *FM LNCS*, vol. 14933, pp. 620–637. Springer (2024). https://doi.org/10.1007/978-3-031-71162-6_32
7. André, É., Bolat, S., Lefauchaux, E., Marinho, D.: stratégFTO: untimed control for timed opacity. In: Artho, C., Ölveczky, P. (eds.) *FTSCS*, pp. 27–33. ACM (2022). <https://doi.org/10.1145/3563822.3568013>
8. André, É., Dufflot-Krémer, M., Laversa, L., Lefauchaux, E.: Execution-time opacity control for timed automata. In: Madeira, A., Knapp, A. (eds.) *SEFM* (2024), to appear
9. André, É., Kryukov, A.: Parametric non-interference in timed automata. In: Li, Y., Liew, A. (eds.) *ICECCS*, pp. 37–42 (2020). <https://doi.org/10.1109/ICECCS51672.2020.00012>
10. André, É., Lefauchaux, E., Lime, D., Marinho, D., Sun, J.: Configuring timing parameters to ensure execution-time opacity in timed automata. In: ter Beek, M.H., Dubslaff, C. (eds.) *TiCSA. Electronic Proceedings in Theoretical Computer Science*, vol. 392, pp. 1–26 (2023). <https://doi.org/10.4204/EPTCS.392.1>, invited paper
11. André, É., Lefauchaux, E., Marinho, D.: Expiring opacity problems in parametric timed automata. In: Ait-Ameur, Y., Khendek, F. (eds.) *ICECCS*, pp. 89–98 (2023). <https://doi.org/10.1109/ICECCS59891.2023.00020>

12. André, É., Lime, D., Marinho, D., Sun, J.: Guaranteeing timed opacity using parametric timed model checking. *ToSEM* **31**(4), 1–36 (2022). <https://doi.org/10.1145/3502851>
13. André, É., Dépernet, S., Lefauchaux, E.: The bright side of timed opacity (extended version). Technical report abs/2408.12240, arXiv, September 2024. <http://arxiv.org/abs/2408.12240>
14. Arcile, J., André, É.: Timed automata as a formalism for expressing security: a survey on theory and practice. *CSUR* **55**(6), 1–36 (2023). <https://doi.org/10.1145/3534967>
15. Barbuti, R., Francesco, N.D., Santone, A., Tesei, L.: A notion of non-interference for timed automata. *FI* **51**(1-2), 1–11 (2002)
16. Barbuti, R., Tesei, L.: A decidable notion of timed non-interference. *FI* **54**(2-3), 137–150 (2003)
17. Benattar, G., Cassez, F., Lime, D., Roux, O.H.: Control and synthesis of non-interferent timed systems. *IJC* **88**(2), 217–236 (2015). <https://doi.org/10.1080/00207179.2014.944356>
18. Bryans, J.W., Koutny, M., Mazaré, L., Ryan, P.Y.A.: Opacity generalised to transition systems. *IseCure* **7**(6), 421–435 (2008). <https://doi.org/10.1007/s10207-008-0058-x>
19. Cassez, F.: The dark side of timed opacity. In: Park, J.H., Chen, H., Atiquzzaman, M., Lee, C., Kim, T., Yeo, S. (eds.) *ISA*. LNCS, vol. 5576, pp. 21–30. Springer (2009). https://doi.org/10.1007/978-3-642-02617-1_3
20. Dima, C.: Real-time automata. *JALC* **6**(1), 3–23 (2001). <https://doi.org/10.25596/jalc-2001-003>
21. Fearnley, J., Jurdzinski, M.: Reachability in two-clock timed automata is PSPACE-complete. *I&C* **243**, 26–36 (2015). <https://doi.org/10.1016/J.IC.2014.12.004>
22. Gardey, G., Mullins, J., Roux, O.H.: Non-interference control synthesis for security timed automata. *ENTCS* **180**(1), 35–53 (2007). <https://doi.org/10.1016/j.entcs.2005.05.046>
23. Klein, J., Kogel, P., Glesner, S.: Verifying opacity of discrete-timed automata. In: Plat, N., Gnesi, S., Furia, C.A., Lopes, A. (eds.) *FormaliSE*, pp. 55–65. ACM (2024). <https://doi.org/10.1145/3644033.3644376>
24. Mazaré, L.: Using unification for opacity properties. In: Ryan, P. (ed.) *WITS*, pp. 165–176, April 2004
25. Ouaknine, J., Worrell, J.: On the language inclusion problem for timed automata: closing a decidability gap. In: *LICS*, pp. 54–63. IEEE Computer Society (2004). <https://doi.org/10.1109/LICS.2004.1319600>
26. Wang, L., Zhan, N.: Decidability of the initial-state opacity of real-time automata. In: Jones, C.B., Wang, J., Zhan, N. (eds.) *Symposium on Real-Time and Hybrid Systems - Essays Dedicated to Professor Chaochen Zhou on the Occasion of His 80th Birthday*, LNCS, vol. 11180, pp. 44–60. Springer (2018). https://doi.org/10.1007/978-3-030-01461-2_3
27. Wang, L., Zhan, N., An, J.: The opacity of real-time automata. *TCAD* **37**(11), 2845–2856 (2018). <https://doi.org/10.1109/TCAD.2018.2857363>
28. Zhang, K.: State-based opacity of labeled real-time automata. *TCS* **987**, 114373 (2024). <https://doi.org/10.1016/J.TCS.2023.114373>



Clock-Dependent Probabilistic Timed Automata with One Clock and No Memory

Jeremy Sproston^(✉) 

University of Turin, Turin, Italy
jeremy.sproston@unito.it

Abstract. Clock-dependent probabilistic timed automata extend probabilistic timed automata by letting the probabilities of discrete transitions depend on the exact values of clock variables. The probabilistic reachability problem for clock-dependent probabilistic timed automata has been shown previously to be undecidable. We consider a subclass with one clock and in which nondeterministic choice is made in a memoryless fashion, i.e., nondeterministic choice depends on the current state only. We show that, for this subclass, the reachability problem can be solved by constructing and analysing a finite-state parametric Markov chain using established methods.

Keywords: Probabilistic model checking · Timed automata · Parametric Markov chains

1 Introduction

The development of complex systems can benefit from automatic verification techniques such as model checking. In a number of application contexts, it is important to reason not just about qualitative aspects of the system (such as reaching an error state, or completing a task) but also quantitative aspects (such as the likelihood of reaching an error state, or whether a task can be completed within a certain deadline). In this paper, we consider probabilistic timed systems, for which the modelling of probabilities and timing aspects of system behaviours is key to their verification. We focus on *probabilistic timed automata* (PTAs) [11, 19, 24], which combine aspects of timed automata [2] (clock variables, constraints on clocks, resets of clocks) and Markov decision processes (MDPs) [25] (transitions are made using a combination of nondeterministic and probabilistic choice), and which have been used to model a number of systems, ranging from network protocols to scheduling problems with uncertainty. Two key characteristics of PTAs are that (1) probabilistic choice is made over the discrete components of the model, rather than over time durations or clock values, and (2) the actual probability values used for probabilistic choices depend only on whether clock values satisfy or not certain clock constraints. Characteristic (2)

has been generalised in the formalism of clock-dependent probabilistic timed automata (cdPTAs) [26], in which relationships between clock values and probabilities used for probabilistic choice can be expressed. Hence cdPTAs are appropriate for modelling systems in which the likelihood of certain events changes over time. For example, for a digital system interacting with humans, the probability of human error may increase over time [10]; in a smart farming system, the greater the time taken to fill a trailer with grain, the higher the probability that a human supervisor regards the trailer to be sufficiently full, but also the higher the probability that the grain will be subsequently ruined by rain during transit (this example adopts elements from [22]). Previous work has showed that the reachability problem for cdPTAs, which asks whether there exists a resolution of nondeterministic choice such that a set of target states is reached with probability at least $\lambda \in (0, 1]$, is undecidable, although upper and lower bounds on reachability probabilities can be computed by analysis of an approximate finite-state MDP [26].

Another approach to the reachability problem for cdPTAs is to consider the construction of a finite-state model that represents exactly the behaviour of a cdPTA belonging to a certain subclass. An exact finite-state construction for the subclass of cdPTAs that feature one clock variable and a requirement specifying that the clock must be reset to 0 between probabilistic choices that depend on the exact value of the clock, thereby guaranteeing the independence of those probabilistic choices, has been presented in [27]. The formalism used for that construction is interval Markov chains [13], in which intervals on probabilities are given for each transition, representing uncertainty with regard to actual probability with which transitions are made. The aforementioned requirement on the independence of non-trivial clock dependencies, as used in [27], does not allow a key characteristic of cdPTAs to be employed, namely the ability to express situations of trade-offs between successive probabilistic choices.

This key characteristic is inherent to the cdPTA of Fig. 1, which models the simple smart farming system described above. We adopt the usual conventions for illustrating cdPTAs: locations are drawn as circles, probability distributions are indicated by black boxes and the boxes' outgoing edges, constraints on the unique clock x label locations (invariant conditions) and edges from locations to black boxes (guard conditions enabling the choice of probability distributions), and expressions determining probabilities of transitions are denoted by grey boxes labelling edges from black boxes to locations (such edges may also feature a clock reset denoted by $\{x\}$). The initial location F represents the trailer being filled with grain, location T represents the trailer being in transit, location S represents the trailer being under shelter at its destination, and location \mathbf{X} represents the failure of the system (either the human supervisor regards the trailer to be insufficiently full or the grain is ruined by rain). The behaviour of the cdPTA takes the following form. On entry to a location, a nondeterministic choice is made regarding the amount of time that elapses while remaining in that location. The value of the clock x increases by the chosen time delay. The choice regarding the amount of time to elapse is constrained by the invariant

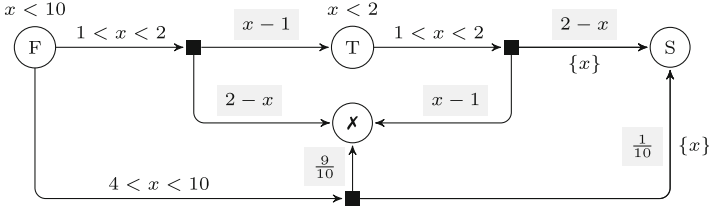


Fig. 1. A 1c-cdPTA modelling a simple task with a trade-off.

condition of the location. After the chosen time delay, if the current value of the clock satisfies the guard condition of one of the location’s outgoing edges, then that edge can be chosen nondeterministically. A probabilistic choice is then made according to the probability distribution that corresponds to the black box that is the target of the chosen edge, where the probabilities regarding the subsequent edge to traverse can depend on the current value of the clock. Traversing edges and black boxes is instantaneous. Observe that in the cdPTA of Fig. 1, when taking the transition to the right of the initial location F, the probability of making a transition to location T increases as the value of the clock x increases (the greater the time dedicated to filling the trailer with grain, the higher the probability that the human supervisor is satisfied); however the probability of subsequently making a transition to location S decreases as the value of the clock x increases (the weather forecast indicates that the probability of the arrival of rain increases until two time units have elapsed). Hence, the maximum probability of reaching location S from F, passing through T, is obtained by letting t units of time elapse in F, then letting no time elapse in location T: we require a framework that takes into account the interdependence of the probability of *both* transitions along the path from F to S through T to reason about the value of t that attains this maximum probability. As a final note, in order to reason about the overall optimal behaviour of the cdPTA, we must also take into account the lowermost edge from location F, which corresponds to filling the trailer completely then waiting until the peak of the storm has passed (after four time units), after which the probability of ruining the grain in transit is $\frac{9}{10}$ regardless of the exact time delay chosen.

In this paper, as in [27], we construct an exact, finite-state abstraction of a cdPTA with one clock (abbreviated as 1c-cdPTA). However, in contrast to [27], we do not impose the restriction that the clock must be reset between the transitions whose probability depends on the exact value of the clock. We use parametric Markov chains (pMCs) [9, 20] as a formalism for the exact, finite-state abstraction. As in interval Markov chains, pMCs represent uncertainty with regard to transition probabilities; however they also allow the expression of dependencies between transition probabilities of different states. In order to obtain a finite-state pMC, we require that the nondeterministic choices made in the 1c-cdPTA are memoryless and finitely-uniform: the underlying infinite state space of the 1c-cdPTA is partitioned into a finite number of equivalence classes

(using the standard notion of regions from the literature of timed automata with one clock [21]), and nondeterministic choices depend on the equivalence class of the current state (rather than on the history of states visited). Our approach is inspired by the precedent of [15] in the context of the use of pMCs for finite-state control of partially-observable MDPs: in both approaches, nondeterministic choices are represented by parameter values, and an instantiation of parameter values corresponds to a finite-state strategy for resolving nondeterministic choice. A novelty of this paper is to establish a relationship between the clock values resulting from the elapse of time and parameters of the pMC. This relationship requires comparison between parameters: in the example of Fig. 1, if the value of the clock x is equal to v when location F is exited, then the value of x when location T is exited must be at least v ; this relationship between clock values is carried over to the parameters of the pMC in order to represent faithfully the 1c-cdPTA. As a consequence of the pMC construction and the results on pMCs of [16], establishing whether there exists a memoryless and finitely-uniform strategy for resolving nondeterministic choice of a 1c-cdPTA such that a set of target states is reached with a probability at least some threshold $\lambda \in (0, 1]$ is in ETR (the complexity class of problems with a polynomial-time many-one reduction to deciding membership in the existential theory of the reals).

Related work. Apart from the references given above, we also mention the following related work. A notion of uniformity of strategies based on a finite partitioning of the state space of a timed automaton game has been presented in [7], and inspired partly our notion of finitely-uniform strategies. Stochastic timed automata [5] and $1\frac{1}{2}$ -player stochastic timed games [1, 6] are variants of PTAs with probabilistic choice over time delays (hence not exhibiting characteristic (1) of PTAs described above).

2 Preliminaries

We use \mathbb{R} to denote the set of real numbers, $\mathbb{R}_{\geq 0}$ to denote the set of non-negative real numbers, \mathbb{Q} to denote the set of rational numbers, and \mathbb{N} to denote the set of natural numbers. A (discrete) probability *distribution* over a countable set Q is a function $\mu : Q \rightarrow [0, 1]$ such that $\sum_{q \in Q} \mu(q) = 1$. Let $\text{Dist}(Q)$ be the set of distributions over Q . For a (possibly uncountable) set Q and a function $\mu : Q \rightarrow [0, 1]$, we define $\text{support}(\mu) = \{q \in Q \mid \mu(q) > 0\}$. Then, for an uncountable set Q , we define $\text{Dist}(Q)$ to be the set of functions $\mu : Q \rightarrow [0, 1]$ such that $\text{support}(\mu)$ is a countable set and μ restricted to $\text{support}(\mu)$ is a distribution. Given a binary function $f : Q \times Q \rightarrow [0, 1]$ and element $q \in Q$, we denote by $f(q, \cdot) : Q \rightarrow [0, 1]$ the unary function such that $f(q, \cdot)(q') = f(q, q')$ for each $q' \in Q$.

Let V be a finite set of real-valued variables called *parameters*. We use $\mathbb{Q}[V]$ to denote the set of rational polynomials over V with coefficients in \mathbb{Q} . An *instantiation* of V is a function $u : V \rightarrow \mathbb{R}$ associating a real value with each parameter in V . Given $f \in \mathbb{Q}[V]$ and instantiation u of V , we denote by $f[u]$ the value obtained from f by substituting each $p \in V$ by $u(p)$. Given the function $g : Q \rightarrow \mathbb{Q}[V]$ and an instantiation u of V , we denote by $g[u] : Q \rightarrow \mathbb{R}$ the

function such that $g[u](q) = g(q)[u]$ for each $q \in Q$. An instantiation u of V is *distribution-inducing* for g if $g[u] \in \text{Dist}(Q)$.

Markov Chains and Markov Decision Processes. A *Markov chain* (MC) \mathcal{C} is a tuple (S, \bar{s}, \mathbf{P}) where S is a set of states with initial state $\bar{s} \in S$, and $\mathbf{P} : S \times S \rightarrow [0, 1]$ is a transition probability function such that $\mathbf{P}(s, \cdot) \in \text{Dist}(S)$ for each state $s \in S$. An (*infinite*) *path* of MC \mathcal{C} is an infinite sequence $s_0 s_1 \cdots$ of states such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. Given a path $\mathbf{r} = s_0 s_1 \cdots$ and $i \geq 0$, we let $\mathbf{r}(i) = s_i$ be the $(i + 1)$ -th state along \mathbf{r} . The set of paths of \mathcal{C} starting in initial state \bar{s} is denoted by $\text{Paths}^{\mathcal{C}}$. Similarly, a finite path of \mathcal{C} is a finite sequence $r = s_0 s_1 \cdots s_n$ such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $0 \leq i < n$. We let $r(i) = s_i$ for $0 \leq i \leq n$ be the $(i + 1)$ -th state along r , and let $\text{Paths}_*^{\mathcal{C}}$ be the set of finite paths of \mathcal{C} starting in initial state \bar{s} . Given a finite path $r = s_0 s_1 \cdots s_{n-1} s_n$ in $\text{Paths}_*^{\mathcal{C}}$, let $\text{Pr}_*^{\mathcal{C}}(r) = \mathbf{P}(s_0, s_1) \cdots \mathbf{P}(s_{n-1}, s_n)$. We can extend uniquely $\text{Pr}_*^{\mathcal{C}}$ to obtain a probability measure $\text{Pr}^{\mathcal{C}}$ over $\text{Paths}^{\mathcal{C}}$ (for more details, see [4]). Given $T \subseteq S$, we define $\diamond T = \{\mathbf{r} \in \text{Paths}^{\mathcal{C}}(\bar{s}) \mid \exists i \in \mathbb{N}. \mathbf{r}(i) \in T\}$ as the set of infinite paths of \mathcal{C} such that some state of T is visited along the path. Hence $\text{Pr}^{\mathcal{C}}(\diamond T)$ denotes the probability that a state in T is visited from the initial state in \mathcal{C} .

Let $\mathcal{R} \subseteq S \times S$ be an equivalence relation on S , and let S/\mathcal{R} be the set of equivalence classes of \mathcal{R} . Given $C \in S/\mathcal{R}$, we let $\mathbf{P}(s, C) = \sum_{s' \in C} \mathbf{P}(s, s')$. An equivalence \mathcal{R} is a *T-preserving probabilistic bisimulation* on \mathcal{C} if $(s, s') \in \mathcal{R}$ implies that (1) $s \in T$ if and only if $s' \in T$, and (2) $\mathbf{P}(s, C) = \mathbf{P}(s', C)$ for each $C \in S/\mathcal{R}$ [23]. Let $\mathcal{C}_1 = (S_1, \bar{s}_1, \mathbf{P}_1)$ and $\mathcal{C}_2 = (S_2, \bar{s}_2, \mathbf{P}_2)$ be two DTMCs such that $S_1 \cap S_2 = \emptyset$, and let $\mathcal{C}_1 \uplus \mathcal{C}_2 = (S_1 \cup S_2 \cup \bar{s}, \bar{\mathbf{P}})$, where (1) $\bar{s} \notin S_1 \cup S_2$, (2) $\mathbf{P}(s, \cdot) = \mathbf{P}_1(s, \cdot)$ if $s \in S_1$ and $\mathbf{P}(s, \cdot) = \mathbf{P}_2(s, \cdot)$ otherwise and (3) $\mathbf{P}(\bar{s}, \bar{s}_1) = \lambda$ and $\mathbf{P}(\bar{s}, \bar{s}_2) = 1 - \lambda$ for some arbitrarily chosen $\lambda \in (0, 1)$. For $T \subseteq S$ and a *T-preserving probabilistic bisimulation* \mathcal{R} on $\mathcal{C}_1 \uplus \mathcal{C}_2$ such that $(\bar{s}_1, \bar{s}_2) \in \mathcal{R}$, we have $\text{Pr}^{\mathcal{C}_1}(\diamond T) = \text{Pr}^{\mathcal{C}_2}(\diamond T)$ [3].

A *Markov decision process* (MDP) $\mathcal{M} = (S, \bar{s}, A, \Delta)$ comprises a set of states S with an initial state $\bar{s} \in S$, a set of actions A , and a probabilistic transition function $\Delta : S \times A \rightarrow \text{Dist}(S) \cup \{\perp\}$. The symbol \perp represents the unavailability of an action in a state, i.e., $\Delta(s, a) = \perp$ signifies that action $a \in A$ is not available in state $s \in S$. For each state $s \in S$, let $A(s) = \{a \in A \mid \Delta(s, a) \neq \perp\}$, and assume that $A(s) \neq \emptyset$, i.e., there is at least one available action in each state. Transitions from state to state of an MDP are performed in two steps: if the current state is s , the first step concerns a nondeterministic selection of an action $a \in A(s)$; the second step comprises a probabilistic choice, made according to the distribution $\Delta(s, a)$, as to which state to make the transition (that is, a transition to a state $s' \in S$ is made with probability $\Delta(s, a)(s')$). In general, the sets of states and actions can be uncountable. We say that an MDP is *finite* if S and A are finite sets. A (n infinite) *path* of an MDP \mathcal{M} is a sequence $s_0 a_0 s_1 a_1 \cdots$ such that $a_i \in A(s_i)$ and $\Delta(s_i, a_i)(s_{i+1}) > 0$ for all $i \geq 0$. Given an infinite path $\mathbf{r} = s_0 a_0 s_1 a_1 \cdots$ and $i \geq 0$, we let $\mathbf{r}(i) = s_i$ be the $(i + 1)$ -th state along \mathbf{r} . Let $\text{Paths}^{\mathcal{M}}$ be the set of infinite paths of \mathcal{M} starting in the initial state \bar{s} . A finite path is a sequence $r = s_0 a_0 s_1 a_1 \cdots a_{n-1} s_n$ such that $a_i \in A(s_i)$ and

$\Delta(s_i, a_i)(s_{i+1}) > 0$ for all $0 \leq i < n$. Let $last(r) = s_n$ denote the final state of r . For $a \in A(s_n)$ and $s \in S$ such that $\Delta(s_n, a)(s) > 0$, we use ras to denote the finite path $s_0 a_0 s_1 a_1 \cdots a_{n-1} s_n a s$. Let $Paths_*^{\mathcal{M}}$ be the set of finite paths of the MDP \mathcal{M} starting in the initial state \bar{s} .

A *strategy* of \mathcal{M} is a function $\sigma : Paths_*^{\mathcal{M}} \rightarrow \bigcup_{s \in S} \text{Dist}(A(s))$ such that $\sigma(r) \in \text{Dist}(A(last(r)))$ and $|\text{support}(\sigma(r))|$ is finite, for all $r \in Paths_*^{\mathcal{M}}$. Let $\Sigma^{\mathcal{M}}$ be the set of strategies of the MDP \mathcal{M} . We say that infinite path $\mathbf{r} = s_0 a_0 s_1 a_1 \cdots$ is *generated* by σ if $\sigma(s_0 a_0 s_1 a_1 \cdots a_{i-1} s_i)(a_i) > 0$ for all $i \in \mathbb{N}$. Let $Paths^\sigma$ be the set of paths generated by σ . The set $Paths_*^\sigma$ of finite paths generated by σ is defined similarly. Given a strategy $\sigma \in \Sigma^{\mathcal{M}}$, we can define a countably infinite-state MC \mathcal{C}^σ , called the *induced MC of σ* , that corresponds to the behaviour of σ : we let $\mathcal{C}^\sigma = (Paths_*^\sigma, \bar{s}, \mathbf{P}^\sigma)$, where, for $r, r' \in Paths_*^\sigma$, we have $\mathbf{P}^\sigma(r, r') = \sigma(r)(a) \cdot \Delta(last(r), a)(s)$ if $r' = ras$ and $a \in A(last(r))$, and $\mathbf{P}^\sigma(r, r') = 0$ otherwise. For simplicity, we write Pr^σ rather than $\text{Pr}^{\mathcal{C}^\sigma}$ for the probability measure associated with \mathcal{C}^σ . Given $T \subseteq S$, and given the 1-to-1 relationship between (finite and infinite) paths of \mathcal{M} generated by σ and paths of \mathcal{C}^σ , we write $\diamond T$ to denote the set of paths of \mathcal{C}^σ that correspond to T being visited in \mathcal{M} ; formally $\diamond T = \{\mathbf{r} \in Paths^{\mathcal{C}^\sigma} \mid \exists i \in \mathbb{N}. last(\mathbf{r}(i)) \in T\}$. Hence $\text{Pr}^\sigma(\diamond T)$ denotes the probability of reaching the set T in \mathcal{M} while following the strategy σ . We consider whether there exists a strategy belonging to a particular set of strategies such that the probability of reaching a certain set of states is at least some threshold. More precisely, given MDP $\mathcal{M} = (S, \bar{s}, A, \Delta)$, strategy set $\Sigma' \subseteq \Sigma^{\mathcal{M}}$, target set $T \subseteq S$ and threshold $\lambda \in (0, 1]$, the (*existential, lower-bounded, non-strict*) *reachability problem for \mathcal{M}, Σ', T and λ* is to decide whether there exists a strategy $\sigma \in \Sigma'$ such that $\text{Pr}^\sigma(\diamond T) \geq \lambda$.

Parametric Markov Chains. A *parametric Markov chain (pMC)* $\mathcal{D} = (S, \bar{s}, V, \Delta)$ comprises a finite set of states S , initial state $\bar{s} \in S$, a finite set of parameters V and a parametric transition function $\Delta : S \times S \rightarrow \mathbb{Q}[V]$. When considering a pMC $\mathcal{D} = (S, \bar{s}, V, \Delta)$, we consider only instantiations of V that are distribution-inducing for $\Delta(s, \cdot)$ for all states $s \in S$. Let $\text{Inst}_{\mathcal{D}}$ be the set of instantiations for \mathcal{D} , and we consider only pMCs for which $\text{Inst}_{\mathcal{D}} \neq \emptyset$. Given instantiation $u \in \text{Inst}_{\mathcal{D}}$, we can observe that $(S, \bar{s}, \Delta[u])$ is an MC, which we denote by $\mathcal{D}[u]$.

We consider the following feasibility problem with respect to reachability for pMCs. Given a pMC $\mathcal{D} = (S, \bar{s}, V, \Delta)$, set of instantiations $\mathbf{U} \subseteq \text{Inst}_{\mathcal{D}}$, target set $T \subseteq S$ and threshold $\lambda \in (0, 1]$, the *feasibility problem for $\mathcal{D}, \mathbf{U}, T$ and λ* is to decide whether there exists an instantiation $u \in \mathbf{U}$ such that $\text{Pr}^{\mathcal{D}[u]}(\diamond T) \geq \lambda$.

3 Clock-Dependent Probabilistic Timed Automata

We now recall the definition of clock-dependent probabilistic timed automata [26], focussing on the subclass with one clock variable [27]. This clock variable will be denoted by x . A *clock valuation* is a value $v \in \mathbb{R}_{\geq 0}$, interpreted as the current value of clock x . Following the usual notational conventions for modelling

formalisms based on timed automata, we use the powerset notation $2^{\{x\}}$ to refer to the set $\{\{x\}, \emptyset\}$, which we will use in the sequel to indicate whether the clock is reset to 0 (denoted by $\{x\}$) or retains its current value (denoted by \emptyset). A *clock constraint* is a conjunction of atomic formulae of the form $x \sim c$, where $\sim \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{N}$. The set of clock constraints is defined as Ψ . A clock valuation v satisfies a clock constraint ψ , denoted by $v \models \psi$, if ψ resolves to true when substituting each occurrence of clock x with v .

Dependencies between clock values and transition probabilities will be expressed using rational polynomials over x . We refer to functions of the form $g : Q \rightarrow \mathbb{Q}[\{x\}]$ as *templates on Q* , and use $\text{Templates}(Q)$ to refer to the set of such functions. Given a clock constraint $\psi \in \Psi$, a template $g \in \text{Templates}(Q)$ is a *distribution template with respect to ψ* if v is distribution-inducing for g for all $v \in \mathbb{R}_{\geq 0}$ such that $v \models \psi$.

A *clock-dependent probabilistic timed automaton with one-clock* (1c-cdPTA) $\mathcal{A} = (L, \bar{l}, \text{inv}, \text{prob})$ comprises the following components:

- a finite set L of locations with an *initial location* $\bar{l} \in L$;
- a function $\text{inv} : L \rightarrow \Psi$ associating an *invariant condition* with each location;
- a set $\text{prob} \subseteq L \times \Psi \times \text{Templates}(2^{\{x\}} \times L)$ of *probabilistic edges*, containing triples of the form (l, \mathbf{g}, \wp) , where l is the source location of the probabilistic edge, \mathbf{g} is a clock constraint called the *guard* of the probabilistic edge, and \wp is a distribution template with respect to $\mathbf{g} \wedge \text{inv}(l)$.

Given a probabilistic edge $\mathbf{p} \in \text{prob}$, we write $\text{src}(\mathbf{p})$, $\text{grd}(\mathbf{p})$ and $\text{tpl}(\mathbf{p})$ for the source location, guard and distribution template of \mathbf{p} , respectively.

The behaviour of a 1c-cdPTA takes a similar form to that of a standard (one-clock) probabilistic timed automaton [11, 17, 19]. A *state* of a 1c-cdPTA is a pair comprising a location and a clock valuation satisfying the location's invariant condition, i.e., $(l, v) \in L \times \mathbb{R}_{\geq 0}$ such that $v \models \text{inv}(l)$. From a state (l, v) , a nondeterministically-chosen amount of time $t \in \mathbb{R}_{\geq 0}$ elapses, increasing the value of the clock to $\tilde{v} = v + t$. The current location's invariant condition $\text{inv}(l)$ must remain satisfied continuously while time passes. A probabilistic edge $(l', \mathbf{g}, \wp) \in \text{prob}$ can then be chosen from state (l, \tilde{v}) if $l = l'$ and the clock constraint \mathbf{g} is satisfied by \tilde{v} . The choice of which such probabilistic edge to take is nondeterministic. Once a probabilistic edge (l', \mathbf{g}, \wp) has been chosen, a successor location, and whether to reset the clock to 0, is chosen probabilistically according to the distribution $\wp[\tilde{v}]$. For example, in the case of the 1c-cdPTA of Fig. 1, from state $(F, 0)$ (i.e., the location is F and the value of clock x is equal to 0), a nondeterministic choice is made as to the amount of time to elapse and which probabilistic edge to select. Consider the case in which delay $t \in (1, 2)$ elapses, increasing the value of x to $\tilde{v} = 0 + t = t$, following which the uppermost probabilistic edge leaving F is traversed. The resulting state is (T, \tilde{v}) with probability $\tilde{v} - 1$, and (X, \tilde{v}) with probability $2 - \tilde{v}$.

In order to simplify the definition of the semantics of 1c-cdPTAs, we make a number of standard assumptions (see [27] for more details). Firstly, invariant conditions bound the clock from above only: for each $l \in L$, the invariant condition $\text{inv}(l)$ is $x \leq c$ for some $c \in \mathbb{N}$, or $x < c$ for some $c \in \mathbb{N} \setminus \{0\}$. Secondly,

the guard of some probabilistic edge can always be satisfied, either in the current state or by letting time elapse. This assumption is expressed by specifying that the guard of some probabilistic edge is satisfied immediately prior to the invariant condition becoming unsatisfied: formally, for each $l \in L$, there exists $\mathbf{p} \in \text{prob}$ with $\text{src}(\mathbf{p}) = l$ and where $\text{grd}(\mathbf{p})$ is such that (1) if $\text{inv}(l) = (x \leq c)$ then $c \models \text{grd}(\mathbf{p})$ (viewing c as a clock valuation), and (2) if $\text{inv}(l) = (x < c)$ then $c - \varepsilon \models \text{grd}(\mathbf{p})$ for all $\varepsilon \in (0, 1)$.¹ Thirdly, target states resulting from probabilistic edges satisfy their invariants: for each $\mathbf{p} \in \text{prob}$, $l \in L$ and $v \in \mathbb{R}_{\geq 0}$ such that $v \models \mathbf{g}$, if $\text{tpl}(\mathbf{p})[v](\emptyset, l) > 0$ then $v \models \text{inv}(l)$.

The semantics of the 1c-cdPTA $\mathcal{A} = (L, \bar{l}, \text{inv}, \text{prob})$ is the MDP $\llbracket \mathcal{A} \rrbracket = (S, \bar{s}, A, \Delta)$ where:

- $S = \{(l, v) \in L \times \mathbb{R}_{\geq 0} \mid v \models \text{inv}(l)\}$;
- $\bar{s} = (\bar{l}, 0)$;
- $A = \mathbb{R}_{\geq 0} \times \text{prob}$;
- for $(l, v) \in S$, $\tilde{v} \in \mathbb{R}_{\geq 0}$ and $\mathbf{p} \in \text{prob}$ such that (1) $\tilde{v} \geq v$, (2) $\tilde{v} \models \text{grd}(\mathbf{p})$ and (3) $w \models \text{inv}(l)$ for all $v \leq w \leq \tilde{v}$, then we let $\Delta((l, v), (\tilde{v}, \mathbf{p}))$ be the distribution such that, for $(l', v') \in S$:

$$\Delta((l, v), (\tilde{v}, \mathbf{p}))((l', v')) = \begin{cases} \text{tpl}(\mathbf{p})[\tilde{v}](\{x\}, l') + \text{tpl}(\mathbf{p})[\tilde{v}](\emptyset, l') & \text{if } v' = \tilde{v} = 0 \\ \text{tpl}(\mathbf{p})[\tilde{v}](\emptyset, l') & \text{if } v' = \tilde{v} > 0 \\ \text{tpl}(\mathbf{p})[\tilde{v}](\{x\}, l') & \text{if } v' = 0, \tilde{v} > 0 \\ 0 & \text{otherwise;} \end{cases}$$

if conditions (1), (2) and (3) are not satisfied, we let $\Delta((l, v), (\tilde{v}, \mathbf{p})) = \perp$.

The summation in the first case of the definition of Δ reflects the fact that, for $v' = \tilde{v} = 0$, for obtaining v' from \tilde{v} , it is immaterial whether the clock is reset.

In the sequel, we consider reachability of a certain set of locations in the 1c-cdPTA. Let $F \subseteq L$ be a set of locations, and let $T_F = \{(l, v) \in S \mid l \in F\}$ be the set of states of $\llbracket \mathcal{A} \rrbracket$ that have their location component in F . Hence, given a strategy σ of $\llbracket \mathcal{A} \rrbracket$, $\text{Pr}^\sigma(\diamond T_F)$ denotes the probability of reaching the set of locations F under the strategy σ .

4 Translation from 1c-cdPTAs to pMCs

Memoryless Strategies for 1c-cdPTAs. Our notion of memoryless strategies for 1c-cdPTAs, which we now present, consists of two aspects: firstly, such strategies depend only on the current state of the 1c-cdPTA; secondly, there exists a finite partition of the state space of the 1c-cdPTA such that, for each class of the partition, strategies behave uniformly over all states in the class. In order to define this partition, we use the notion of regions² for timed automata

¹ Note that the interval $(0, 1)$ in condition (2) can be replaced by $(0, \lambda)$ for any $\lambda \in (0, 1)$, because either all valuations in $(c - 1, c)$ satisfy $\text{grd}(\mathbf{p})$ or none do.

² Note that sets of instantiations are often referred to as regions in the pMC literature (for example, in [14]). Instead, in this paper, we adopt the notion of regions from the timed automata literature.

with one clock [21]. Then the behaviour (in terms of which clock value to let time elapse to, and which probabilistic edge to take) of a memoryless strategy is described as being the same for all states of each of the aforementioned regions. The presence of such uniformity means that we are able to define a memoryless strategy of a 1-cdPTA using a finite framework.

Let $\mathcal{A} = (L, \bar{l}, \text{inv}, \text{prob})$ be a 1c-cdPTA. Let $\text{Cst}(\mathcal{A})$ be the set of constants that are used in the guards of probabilistic edges and invariants of \mathcal{A} , and let $\mathbb{B} = \text{Cst}(\mathcal{A}) \cup \{0\}$. We write $\mathbb{B} = \{b_0, b_1, \dots, b_k\}$, where $0 = b_0 < b_1 < \dots < b_k$. The set \mathbb{B} defines the set of intervals $\mathcal{I}_{\mathbb{B}} = \{[b_0, b_0], (b_0, b_1), [b_1, b_1], \dots, [b_k, b_k], [b_k, \infty)\}$, i.e., $\mathcal{I}_{\mathbb{B}}$ is a partition of $\mathbb{R}_{\geq 0}$ into subintervals with endpoints in $\mathbb{B} \cup \{\infty\}$, and where each element of \mathbb{B} has a corresponding closed interval in $\mathcal{I}_{\mathbb{B}}$ comprising only that element. We define a total order on $\mathcal{I}_{\mathbb{B}}$ in the following way: $[b_0, b_0] < (b_0, b_1) < [b_1, b_1] < \dots < [b_k, b_k] < [b_k, \infty)$. Let ψ be a guard of a probabilistic edge or an invariant of \mathcal{A} . By definition, for each $B \in \mathcal{I}_{\mathbb{B}}$, either $B \subseteq \{v \in \mathbb{R}_{\geq 0} \mid v \models \psi\}$ or $B \cap \{v \in \mathbb{R}_{\geq 0} \mid v \models \psi\} = \emptyset$ (either all valuations in B satisfy ψ or none do). We write $B \models \psi$ in the case of $B \subseteq \{v \in \mathbb{R}_{\geq 0} \mid v \models \psi\}$.

The set of *regions* of \mathcal{A} is defined as $\text{Regs} = \{(l, B) \in L \times \mathcal{I}_{\mathbb{B}} \mid B \models \text{inv}(l)\}$. In the sequel, we will refer to a state (l, v) of $\llbracket \mathcal{A} \rrbracket$ as belonging to a region (l', B) if $l = l'$ and $v \in B$. Given a region (l, B) , the set of *region successor actions* of (l, B) , denoted by $\text{RegA}(l, B)$, is defined as the set of pairs where the first element of the pair is an interval \tilde{B} that can be obtained from B by letting time elapse (provided that \tilde{B} does not exceed the upper bound enforced by the invariant condition), and where the second element is a probabilistic edge with source location l and guard which is satisfied by \tilde{B} . Formally, we let:

$$\text{RegA}(l, B) = \{(\tilde{B}, \mathbf{p}) \in \mathcal{I}_{\mathbb{B}} \times \text{prob} \mid \text{src}(\mathbf{p}) = l \wedge B \leq \tilde{B} \wedge \tilde{B} \models (\text{grd}(\mathbf{p}) \wedge \text{inv}(l))\}.$$

We now define, in two steps, our notion of memoryless strategies for 1c-cdPTAs in two steps. First, we introduce region-based controllers which, given a current region of the 1c-cdPTA, specify a distribution over region successor actions, and, for each such action, also determine exactly which clock valuation is attained by letting time elapse. An *action mapping* $\text{act} : \text{Regs} \rightarrow \text{Dist}(\mathcal{I}_{\mathbb{B}} \times \text{prob})$ for \mathcal{A} is a function such that, for each $(l, B) \in \text{Regs}$ and $(\tilde{B}, \mathbf{p}) \in \mathcal{I}_{\mathbb{B}} \times \text{prob}$, if $\text{act}(l, B)(\tilde{B}, \mathbf{p}) > 0$ then $(\tilde{B}, \mathbf{p}) \in \text{RegA}(l, B)$. A *time-elapse valuation mapping* $\text{val} : \text{Regs} \times \mathcal{I}_{\mathbb{B}} \times \text{prob} \rightarrow \mathbb{R}_{\geq 0}$ for \mathcal{A} is a function such that the following conditions are satisfied for each $(l, B) \in \text{Regs}$ and $(\tilde{B}, \mathbf{p}) \in \mathcal{I}_{\mathbb{B}} \times \text{prob}$:

- $\text{val}((l, B), \tilde{B}, \mathbf{p}) \in \tilde{B}$ (the time-elapse valuation obtained from val belongs to time-elapse interval \tilde{B});
- for each $l' \in L$ and $(\tilde{B}', \mathbf{p}') \in \mathcal{I}_{\mathbb{B}} \times \text{prob}$, if:
 - $\text{tpl}(\mathbf{p})[\text{val}((l, B), \tilde{B}, \mathbf{p})](\emptyset, l') > 0$ (the probability of passing from (l, B) to location l' using region successor action (\tilde{B}, \mathbf{p}) while *not* resetting the clock is positive), and
 - $\text{act}(l, B)(\tilde{B}, \mathbf{p}) > 0$ and $\text{act}(l', \tilde{B}')(\tilde{B}', \mathbf{p}') > 0$ (act specifies that (\tilde{B}, \mathbf{p}) can be chosen from (l, B) and $(\tilde{B}', \mathbf{p}')$ can be chosen from (l', \tilde{B}')),

then $\mathbf{val}((l, B), \tilde{B}, \mathbf{p}) \leq \mathbf{val}((l', \tilde{B}), \tilde{B}', \mathbf{p}')$ (\mathbf{val} cannot specify that the clock's value decreases between successive transitions).

Then a *region-based controller* for \mathcal{A} is defined as a pair $(\mathbf{act}, \mathbf{val})$ comprising an action mapping \mathbf{act} and a time-elapse valuation mapping \mathbf{val} .

We can now use a region-based controller $(\mathbf{act}, \mathbf{val})$ to define an associated strategy $\sigma^{\mathbf{act}, \mathbf{val}}$ for the 1c-cdPTA \mathcal{A} . Given a clock valuation $v \in \mathbb{R}_{\geq 0}$, we denote by $\langle v \rangle$ the unique interval in $\mathcal{I}_{\mathbb{B}}$ that contains v . Let $r \in \mathit{Paths}_{*}^{\llbracket \mathcal{A} \rrbracket}$ be a finite path of $\llbracket \mathcal{A} \rrbracket$ and let $(\tilde{v}, \mathbf{p}) \in \mathbb{R}_{\geq 0} \times \mathit{prob}$. We will use (l, v) to denote $\mathit{last}(r)$. First note from the definition of the probabilistic transition function Δ of $\llbracket \mathcal{A} \rrbracket$ and RegA , if $\Delta(\tilde{v}, \mathbf{p}) \neq \perp$, then $(\langle \tilde{v} \rangle, \mathbf{p}) \in \mathit{RegA}(l, B)$. Then we let:

$$\sigma^{\mathbf{act}, \mathbf{val}}(r)(\tilde{v}, \mathbf{p}) = \begin{cases} \mathbf{act}(l, \langle v \rangle)(\langle \tilde{v} \rangle, \mathbf{p}) & \text{if } \tilde{v} = \mathbf{val}((l, \langle v \rangle), \langle \tilde{v} \rangle, \mathbf{p}); \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\sigma^{\mathbf{act}, \mathbf{val}}$ is memoryless: its behaviour after a finite path r depends on the region containing the final state of r . A strategy $\sigma \in \Sigma^{\llbracket \mathcal{A} \rrbracket}$ is a *region-based (memoryless) strategy* if there exists a region-based controller $(\mathbf{act}, \mathbf{val})$ such that $\sigma = \sigma^{\mathbf{act}, \mathbf{val}}$. Let $\Sigma^{\mathit{rbc}} \subseteq \Sigma^{\llbracket \mathcal{A} \rrbracket}$ be the set of region-based strategies of $\llbracket \mathcal{A} \rrbracket$. The *region-based controller reachability problem* for \mathcal{A} , $F \subseteq L$ and $\lambda \in (0, 1]$ is to determine whether there exists a region-based controller $(\mathbf{act}, \mathbf{val})$ for \mathcal{A} such that $\Pr^{\sigma^{\mathbf{act}, \mathbf{val}}}(\diamond T_F) \geq \lambda$. Note that this problem is equivalent to the reachability problem for $\llbracket \mathcal{A} \rrbracket$, Σ^{rbc} , T_F and λ .

Translation from 1c-cdPTAs to pMCs. We now describe how a pMC corresponding to the 1c-cdPTA \mathcal{A} can be defined, and show how feasibility analysis of this pMC can be used to establish the existence of a strategy that is obtained from a region-based controller and that satisfies a reachability property. The intuition is that the states of the pMC are regions of the 1-cdPTA, and that two parameters are used for each region $(l, B) \in \mathit{Regs}$ and region successor action $(\tilde{B}, \mathbf{p}) \in \mathit{RegA}(l, B)$, where the first parameter $p_{(\tilde{B}, \mathbf{p})}^{(l, B)}$ refers to the probability with which (\tilde{B}, \mathbf{p}) is taken from states in (l, B) , and the second parameter $q_{(\tilde{B}, \mathbf{p})}^{(l, B)}$ refers to the clock valuation in \tilde{B} attained after letting time elapse when making a transition from a state in (l, B) using the probabilistic edge \mathbf{p} . The correspondence between the first set of parameters described above and actions mappings, and the second set of parameters with time-elapse valuation mappings, will allow us to use the pMC to answer the region-based controller reachability problem introduced above.

Let $\mathcal{A} = (L, \bar{l}, \mathit{inv}, \mathit{prob})$ be a 1c-cdPTA. The pMC *induced from* \mathcal{A} is defined as $\mathcal{D}^{\mathcal{A}} = (\mathit{Regs}, (\bar{l}, [0, 0]), V^{\mathcal{A}}, \mathbf{\Delta}^{\mathcal{A}})$, where:

- $V^{\mathcal{A}} = \{p_{(\tilde{B}, \mathbf{p})}^{(l, B)}, q_{(\tilde{B}, \mathbf{p})}^{(l, B)} \mid (l, B) \in \mathit{Regs}, (\tilde{B}, \mathbf{p}) \in \mathit{RegA}(l, B)\}$ is the set of parameters;

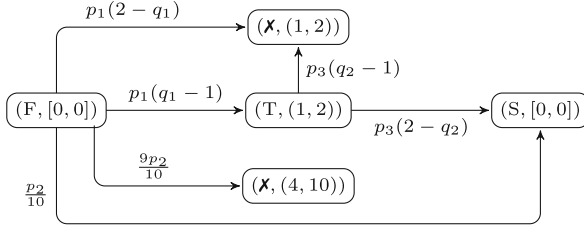


Fig. 2. The pMC corresponding to the 1c-cdPTA of Fig. 1.

- the parametric transition function $\Delta^{\mathcal{A}}$ is such that, for $(l, B), (l', B') \in S^{\mathcal{A}}$, we have:

$$\Delta^{\mathcal{A}}((l, B), (l', B')) = \sum_{(\tilde{B}, \mathbf{p}) \in \text{RegA}(l, B)} \hat{\Delta}^{\mathcal{A}}((l, B), (\tilde{B}, \mathbf{p}), (l', B')),$$

where, for $(\tilde{B}, \mathbf{p}) \in \text{RegA}(l, B)$, we have that $\hat{\Delta}^{\mathcal{A}}((l, B), (\tilde{B}, \mathbf{p}), (l', B'))$ equals:

$$\begin{cases} p_{(\tilde{B}, \mathbf{p})}^{(l, B)} \cdot (\text{tpl}(\mathbf{p})[0](\{x\}, l') + \text{tpl}(\mathbf{p})[0](\emptyset, l')) & \text{if } \tilde{B} = B' = [0, 0], \\ p_{(\tilde{B}, \mathbf{p})}^{(l, \tilde{B})} \cdot \text{tpl}(\mathbf{p})[q_{\tilde{B}, \mathbf{p}}^{(l, B)}](\{x\}, l') & \text{if } \tilde{B} > [0, 0], B' = [0, 0], \\ p_{(\tilde{B}, \mathbf{p})}^{(l, \tilde{B})} \cdot \text{tpl}(\mathbf{p})[q_{\tilde{B}, \mathbf{p}}^{(l, \tilde{B})}](\emptyset, l') & \text{if } \tilde{B} = B' > [0, 0], \\ 0 & \text{otherwise.} \end{cases}$$

Observe that, given regions $(l, B), (l', B') \in \text{Regs}$ and region successor action $(\tilde{B}, \mathbf{p}) \in \text{RegA}(l, B)$, the expression $\hat{\Delta}^{\mathcal{A}}((l, B), (\tilde{B}, \mathbf{p}), (l', B'))$ is a rational polynomial over $V^{\mathcal{A}}$ describing the probability of making a translation from region (l, B) to region (l', B') via region successor action (\tilde{B}, \mathbf{p}) .

In Fig. 2, we show the pMC corresponding to the 1c-cdPTA of Fig. 1. In the following, we use \mathbf{p} (\mathbf{p}' , \mathbf{p}'' , respectively) to refer to upper-left (upper-right, bottom, respectively) probabilistic edge of Fig. 1. Parameter p_1 corresponds to $p_{((1,2), \mathbf{p})}^{(F, [0,0])}$, whereas p_2 corresponds to $p_{((4,10), \mathbf{p}'')}^{(F, [0,0])}$, and p_3 corresponds to $p_{((1,2), \mathbf{p}')}^{(T, (1,2))}$. Furthermore q_1 refers to $q_{((1,2), \mathbf{p})}^{(F, [0,0])}$, and q_2 refers to $q_{((1,2), \mathbf{p})}^{(T, (1,2))}$. For completeness, we

note that the construction of the pMC includes also the parameter $q_{((4,10), \mathbf{p}'')}^{(F, [0,0])}$; however, given that the probability of the transitions of probabilistic edge \mathbf{p}'' do not depend on the value of x , the parameter $q_{((4,10), \mathbf{p}'')}^{(F, [0,0])}$ has no role in the expressions on the transitions from $(F, [0, 0])$ to $(\mathbf{X}, (4, 10))$ and $(S, [0, 0])$ in the pMC.

Given that the role of the pMC $\mathcal{D}^{\mathcal{A}}$ is to represent faithfully the behaviour of the 1c-cdPTA \mathcal{A} , we henceforth consider instantiations that satisfy the following constraints, which reflect closely the constraints imposed on the time-elapse valuation mapping for region-based controllers. An instantiation $u \in \text{Inst}_{\mathcal{D}^{\mathcal{A}}}$ is *time-elapse preserving* if, for each region $(l, B) \in \text{Regs}$ and region successor action $(\tilde{B}, \mathbf{p}) \in \text{RegA}(l, B)$, (1) $q_{(\tilde{B}, \mathbf{p})}^{(l, B)} \in \tilde{B}$, and (2) for each $l' \in L$ and

$(\tilde{B}', \mathbf{p}') \in \text{RegA}(l', \tilde{B})$, if $\text{tpl}(\mathbf{p})[q_{(\tilde{B}, \mathbf{p})}^{(l, B)}](\emptyset, l') > 0$ then $q_{(\tilde{B}, \mathbf{p})}^{(l, B)} \leq q_{(\tilde{B}', \mathbf{p}')}^{(l', \tilde{B})}$. We write $\mathbf{U}^{\mathcal{A}}$ to denote the set of time-elapse preserving instantiations for $\mathcal{D}^{\mathcal{A}}$, and will henceforth consider the feasibility problem for $\mathcal{D}^{\mathcal{A}}$ and $\mathbf{U}^{\mathcal{A}}$.

Correctness of the Translation. Let $F \subseteq L$, let $\text{Reg}_F = \{(l, B) \in \text{Regs} \mid l \in F\}$ be the set of states of $\mathcal{D}^{\mathcal{A}}$ that have their location component in F , and let $\lambda \in (0, 1]$. Our aim is now to show that answers to the feasibility problem for the pMC $\mathcal{D}^{\mathcal{A}}$, set of instantiations $\mathbf{U}^{\mathcal{A}}$, target set Reg_F and threshold λ , and to the region-based controller reachability problem for \mathcal{A} , F and λ coincide. To do this, by definition of these two problems, we require that there exists a region-based strategy $\sigma \in \Sigma^{\text{rbc}}$ such that $\text{Pr}^\sigma(\diamond T_F) \geq \lambda$ if and only if there exists a time-elapse preserving instantiation $u \in \mathbf{U}^{\mathcal{A}}$ such that $\text{Pr}^{\mathcal{D}^{\mathcal{A}}[u]}(\diamond \text{Reg}_F) \geq \lambda$. This can in turn be established by the following result.

Proposition 1. (1) For each region-based strategy $\sigma \in \Sigma^{\text{rbc}}$, there exists a time-elapse preserving instantiation $u \in \mathbf{U}^{\mathcal{A}}$ such that $\text{Pr}^\sigma(\diamond T_F) = \text{Pr}^{\mathcal{D}^{\mathcal{A}}[u]}(\diamond \text{Reg}_F)$. (2) For each time-elapse preserving instantiation $u \in \mathbf{U}^{\mathcal{A}}$, there exists a region-based strategy $\sigma \in \Sigma^{\text{rbc}}$ such that $\text{Pr}^\sigma(\diamond T_F) = \text{Pr}^{\mathcal{D}^{\mathcal{A}}[u]}(\diamond \text{Reg}_F)$.

Proof (sketch). The proof of Proposition 1 relies on establishing a correspondence between a region-based controller associated with a region-based strategy of $\llbracket \mathcal{A} \rrbracket$ and a time-elapse preserving instantiation of $\mathcal{D}^{\mathcal{A}}$. For part (1), given a region-based strategy $\sigma \in \Sigma^{\text{rbc}}$ and an associated region-based controller (act, val) , we define the time-elapse preserving instantiation $u^{\text{act}, \text{val}} \in \mathbf{U}^{\mathcal{A}}$ as follows: for each $(l, B) \in \text{Regs}$ and $(\tilde{B}, \mathbf{p}) \in \text{RegA}(l, B)$, let $u^{\text{act}, \text{val}}(p_{(\tilde{B}, \mathbf{p})}^{(l, B)}) = \text{act}(l, B)(\tilde{B}, \mathbf{p})$ and $u^{\text{act}, \text{val}}(q_{(\tilde{B}, \mathbf{p})}^{(l, B)}) = \text{val}((l, B), \tilde{B}, \mathbf{p})$. Note that $u^{\text{act}, \text{val}}$ is time-elapse preserving due to the similarity between the definitions of region-based controllers and time-elapse preserving instantiations. Similarly, for part (2), given the time-elapse preserving instantiation $u \in \mathbf{U}^{\mathcal{A}}$, we define a region-based controller $(\text{act}_u, \text{val}_u)$ by letting $\text{act}_u(l, B)(\tilde{B}, \mathbf{p}) = u(p_{(\tilde{B}, \mathbf{p})}^{(l, B)})$ and $\text{val}_u((l, B), \tilde{B}, \mathbf{p}) = u(q_{(\tilde{B}, \mathbf{p})}^{(l, B)})$ for each $(l, B) \in \text{Regs}$ and $(\tilde{B}, \mathbf{p}) \in \text{RegA}(l, B)$, hence obtaining the region-based strategy $\sigma^{\text{act}_u, \text{val}_u} \in \Sigma^{\text{rbc}}$. We also note that the fact that u is time-elapse preserving instantiation guarantees that $(\text{act}_u, \text{val}_u)$ satisfies the conditions of the definition of a region-based controller. It remains to justify that these constructions establish that the reachability probabilities considered in Proposition 1 are equal.

Let $\tilde{B} \in \mathcal{I}_{\mathbb{B}}$ and (act, val) be a region-based controller. Then the set

$$\text{Vals}(\tilde{B}) = \{\text{val}((l, B), \tilde{B}', \mathbf{p}) \mid (l, B) \in \text{Regs}, (\tilde{B}', \mathbf{p}) \in \text{RegA}(l, B) \text{ s.t. } \tilde{B}' = \tilde{B}\}$$

of possible time-elapse valuations induced by val and belonging to \tilde{B} is finite. Recalling that, on traversing a probabilistic edge, the value of clock x is either reset to 0 or retains its value from before the traversal, we can conclude that the set of states reached by the region-based strategy $\sigma^{\text{act}, \text{val}}$ is finite, and that we only need to consider finite paths ending in states of $\llbracket \mathcal{A} \rrbracket$ that have valuations

in $\bigcup_{B \in \mathcal{I}_B} \text{Vals}(B)$. Now we establish a relationship between the probability of transitions of the MC induced by a region-based strategy and transitions of the pMC with the associated time-elapse preserving instantiation.

Lemma 1. (1) Let $\sigma \in \Sigma^{\text{rbc}}$ with associated region-based controller (act, val) , let $(l, v) \in S$ be such that $v \in \text{Vals}(\llbracket v \rrbracket)$, let $(\tilde{v}, \mathbf{p}) \in A(l, v)$, and let $(l', v') \in S$. Then $\mathbf{P}^\sigma((l, v), (l, v)(\tilde{v}, \mathbf{p})(l', v')) = \hat{\Delta}^A[u^{\text{act}, \text{val}}((l, \llbracket v \rrbracket), (\llbracket \tilde{v} \rrbracket, \mathbf{p}), (l', \llbracket v' \rrbracket))]$. (2) Let $u \in \mathcal{U}^A$ be a time-elapse preserving instantiation, let $(l, B), (l', B') \in \text{Regs}$, let $v \in \text{Vals}(B)$, and let $(\tilde{B}, \mathbf{p}) \in \text{RegA}(l, B)$. Then $\hat{\Delta}^A[u((l, B), (\tilde{B}, \mathbf{p}), (l', B'))]$ equals:

$$\begin{cases} \mathbf{P}^{\sigma^{\text{act}_u, \text{val}_u}}((l, v), (l, v)(\text{val}_u((l, B), \tilde{B}, \mathbf{p}), \mathbf{p})(l', 0)) & \text{if } B' = [0, 0] \\ \mathbf{P}^{\sigma^{\text{act}_u, \text{val}_u}}((l, v), (l, v)(\text{val}_u((l, B), \tilde{B}, \mathbf{p}), \mathbf{p})(l', \text{val}_u((l, B), \tilde{B}, \mathbf{p}))) & \text{otherwise.} \end{cases}$$

The proof of Lemma 1 relies on the definitions of $u^{\text{act}, \text{val}}$ (for part (1)) and of $\sigma^{\text{act}_u, \text{val}_u}$ (for part (2)) given above in this section. Note that we considered finite paths of at length one and two in the lemma; this is not a restriction, given that region-based strategies are memoryless. Using parts (1) and (2) of Lemma 1, and recalling the result of [3] (presented in Sect. 2) that specifies that probabilistically bisimilar states exhibit the same probabilities of reaching a certain set of states, we can then show the respective parts of Proposition 1 by showing that the initial states of the MCs under consideration ($\mathcal{C}^{\sigma^{\text{act}, \text{val}}}$ and $\mathcal{D}[u^{\text{act}, \text{val}}]$ for part (1), and $\mathcal{C}^{\sigma^{\text{act}_u, \text{val}_u}}$ and $\mathcal{D}[u]$ for part (2), respectively) are related by a $T_F \cup \text{Reg}_F$ -preserving probabilistic bisimulation.

The following corollary is a consequence of Proposition 1 and the definitions of the region-based controller reachability problem and the feasibility problem, and the fact that the latter is in ETR [16].

Corollary 1. *The region-based controller reachability problem is in ETR.*

5 Conclusion

We have presented a method for constructing a finite-state pMC that can be used for solving reachability problems for 1c-cdPTAs when restricted to region-based, memoryless strategies. To our knowledge, tool support for our approach is not available currently: both PRISM [18] and STORM [12] do not at present perform parametric analysis in the case of comparisons between parameters such as $p_1 \leq p_2$, which are required to represent time-preserving instantiations. Future work will consider the extension of the results to the case of finite-memory strategies, using an adaptation of the unfolding approach of [15]. An important question to be answered is whether finite-memory strategies suffice to obtain optimal reachability probabilities for 1c-cdPTAs. Furthermore, recalling that we considered the problem of determining whether a target set can be reached with probability at least λ , we would also like to consider related problems asking whether a target set is reached with probability strictly greater than λ , at most λ , or strictly less than λ . We note that, for the strictly-less-than- λ case, finite-state

strategies do not suffice, a fact that can be obtained by adapting an example from [28] to the 1c-cdPTA context. Finally, while our results show that our reachability problems for 1c-cdPTAs with region-based, memoryless strategies are in the complexity class ETR, the lower bound for our problems is unknown. The ETR lower bound for (quantitative) reachability problems of pMCs [8, 16] relies on a construction which is not directly applicable in our setting, given that 1c-cdPTAs can express only dependencies between clock values in successive states.

Acknowledgments. This work was partly supported by the European Union – Next-GenerationEU – National Recovery and Resilience Plan (NRRP) – MISSION 4 COMPONENT 2, INVESTMENT N. 1.1, CALL PRIN 2022 D.D. 104 02-02-2022 – MEDICA Project, CUP N.D53D23008800006, and the National Recovery and Resilience Plan (NRRP), partnership on “Telecommunications of the Future” (PE0000001 - program “RESTART”, project Net4Future).

References

1. Akshay, S., Bouyer, P., Krishna, S.N., Manasa, L., Trivedi, A.: Stochastic timed games revisited. In: Proceedings of MFCS 2016). *LIPICs*, vol. 58, pp. 8:1–8:14. Leibniz-Zentrum für Informatik (2016). <https://doi.org/10.4230/LIPICs.MFCS.2016.8>
2. Alur, R., Dill, D.L.: A theory of timed automata. *Theoret. Comput. Sci.* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
3. Aziz, A., Singhal, V., Balarin, F., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: It usually works: the temporal logic of stochastic systems. In: Wolper, P. (ed.) *Computer Aided Verification*, pp. 155–165. Springer, Berlin, Heidelberg (1995). https://doi.org/10.1007/3-540-60045-0_48
4. Baier, C., Katoen, J.P.: *Principles of Model Checking*. MIT Press (2008)
5. Bertrand, N., Bouyer, P., Brihaye, T., Menet, Q., Baier, C., Größer, M., Jurdzinski, M.: Stochastic timed automata. *Logical Methods Comput. Sci.* **10**(4) (2014). [https://doi.org/10.2168/LMCS-10\(4:6\)2014](https://doi.org/10.2168/LMCS-10(4:6)2014)
6. Bouyer, P., Forejt, V.: Reachability in stochastic timed games. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikolettseas, S., Thomas, W. (eds.) *Automata, Languages and Programming*, pp. 103–114. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02930-1_9
7. Chatterjee, K., Henzinger, T.A., Prabhu, V.S.: Timed parity games: complexity and robustness. *Logical Methods in Comput. Sci.* **7**(4) (2011). [https://doi.org/10.2168/LMCS-7\(4:8\)2011](https://doi.org/10.2168/LMCS-7(4:8)2011)
8. Chonev, V.: Reachability in augmented interval Markov chains. In: Filiot, E., Jungers, R., Potapov, I. (eds.) *Reachability Problems: 13th International Conference, RP 2019, Brussels, Belgium, September 11–13, 2019, Proceedings*, pp. 79–92. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-30806-3_7
9. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Proceedings of ICTAC 2004. *LNCS*, vol. 3407, pp. 280–294. Springer (2004). https://doi.org/10.1007/978-3-540-31862-0_21
10. Feng, L., Wiltzsche, C., Humphrey, L.R., Topcu, U.: Synthesis of human-in-the-loop control protocols for autonomous systems. *IEEE Trans. Autom. Sci. Eng.* **13**(2), 450–462 (2016). <https://doi.org/10.1109/TASE.2016.2530623>

11. Gregersen, H., Jensen, H.E.: Formal Design of Reliable Real Time Systems. Master's thesis, Department of Mathematics and Computer Science, Aalborg University (1995)
12. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. *Softw. Tools Technol. Transfer* **24**(4) (2022). <https://doi.org/10.1007/S10009-021-00633-Z>
13. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: Proceedings of LICS 1991, pp. 266–277. IEEE Computer Society (1991). <https://doi.org/10.1109/LICS.1991.151651>
14. Junges, S., et al.: Parameter synthesis for Markov models: covering the parameter space. *Formal Methods Syst. Design* **62**, 181–259 (2024). <https://doi.org/10.1007/S10703-023-00442-X>
15. Junges, S., et al.: Finite-state controllers of POMDPs using parameter synthesis. In: Proceedings of UAI 2018, pp. 519–529. AUAI Press (2018)
16. Junges, S., Katoen, J.P., Pérez, G.A., Winkler, T.: The complexity of reachability in parametric Markov decision processes. *J. Comput. Syst. Sci.* **119**, 183–210 (2021). <https://doi.org/10.1016/J.JCSS.2021.02.006>
17. Jurdziński, M., Laroussinie, F., Sproston, J.: Model checking probabilistic timed automata with one or two clocks. *Logical Methods Comput. Sci.* **4**(3) (2008). [https://doi.org/10.2168/LMCS-4\(3:12\)2008](https://doi.org/10.2168/LMCS-4(3:12)2008)
18. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) *Computer Aided Verification*, pp. 585–591. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
19. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. *Theoret. Comput. Sci.* **286**, 101–150 (2002). [https://doi.org/10.1016/S0304-3975\(01\)00046-9](https://doi.org/10.1016/S0304-3975(01)00046-9)
20. Lanotte, R., Maggiolo-Schettini, A., Troina, A.: Parametric probabilistic transition systems for system design and analysis. *Formal Aspects Comput.* **19**(1), 93–109 (2007). <https://doi.org/10.1007/S00165-006-0015-2>
21. Laroussinie, F., Markey, N., Schnoebelen, P.: Model checking timed automata with one or two clocks. In: Gardner, P., Yoshida, N. (eds.) *CONCUR 2004 - Concurrency Theory*, pp. 387–401. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_25
22. Larsen, K.G.: Synthesis of safe and optimal strategies for cyber-physical systems. <https://movep2022.cs.aau.dk/slides/KimGuldstrandLarsen-movep2022.pdf> (2022). Slides presented at MOVEP 2022
23. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. *Inf. Comput.* **94**(1), 1–28 (1991). [https://doi.org/10.1016/0890-5401\(91\)90030-6](https://doi.org/10.1016/0890-5401(91)90030-6)
24. Norman, G., Parker, D., Sproston, J.: Model checking for probabilistic timed automata. *Formal Methods Syst. Design* **43**(2), 164–190 (2013). <https://doi.org/10.1007/S10703-012-0177-X>
25. Puterman, M.L.: *Markov Decision Processes*. J. Wiley & Sons (1994)
26. Sproston, J.: Probabilistic timed automata with clock-dependent probabilities. *Fund. Inform.* **178**(1–2), 101–138 (2021). <https://doi.org/10.3233/FI-2021-2000>
27. Sproston, J.: Probabilistic timed automata with one clock and initialised clock-dependent probabilities. *Logical Methods Comput. Sci.* **17**(4) (2021). [https://doi.org/10.46298/LMCS-17\(4:6\)2021](https://doi.org/10.46298/LMCS-17(4:6)2021)
28. Sproston, J.: Qualitative reachability for open interval Markov chains. *PeerJ Comput. Sci.* **9**, e1489 (2023). <https://doi.org/10.7717/peerj-cs.1489>



Efficient State Estimation of Discrete-Timed Automata

Julian Klein^(✉), Paul Kogel, and Sabine Glesner

Software and Embedded Systems Engineering, TU Berlin, Berlin, Germany
{j.klein,p.kogel,sabine.glesner}@tu-berlin.de

Abstract. State estimation is a fundamental method in control theory that has applications in privacy, fault diagnosis, and the verification of other state inference properties. State estimation methods for timed automata rely on discretizing time into atomic steps. These discrete time steps are enumerated in separate states, significantly limiting scalability for large, realistic systems. In this paper, we propose a more efficient state estimation method for discrete-timed automata. The key idea of our approach is to avoid the enumeration of single time steps when possible. We provide a formal definition of a new state estimator model and an efficient algorithmic approach to derive it from discrete-timed automata. We validate our method on 11 realistic case studies and show a significant decrease in computational costs.

Keywords: State Estimation · Observers · Timed Automata

1 Introduction

With the growing size and complexity of *real-time systems* (RTS), scalable verification is becoming essential. Such verification methods often require estimators [15, 27, 29]. An estimator is a deterministic representation of a target system that provides the set of all possibly active states for any given input [25]. *Deterministic tick automata* (τ -DFA) [9] are an expressive class of estimators that can represent various types of RTS [9, 10, 12, 17]. τ -DFA are standard *deterministic finite automata* (DFA) that enumerate discrete time steps with a special *tick*-symbol τ . The computation of τ -DFA therefore requires the evaluation of every single time step of the target RTS. This limits scalability and makes verification of large real-world systems infeasible in practice.

In this work, we address this lack of scalability with a new state estimation method. The key idea of our method is to avoid evaluating every single time step. Instead, we identify and evaluate only significant time steps (thresholds) that are necessary for the computation of an estimator.

To facilitate this, we make two main contributions. First, we introduce *threshold estimators* (TE), a new class of estimators that is equally expressive as τ -DFA but can be computed more efficiently. Second, we provide an algorithmic approach to derive TE from discrete-timed automata (TA), a powerful and extensively studied model for RTS [3, 6].

We evaluate our method on 11 diverse case studies from the domains of network protocols and architectures, automotive systems, and smart home devices. Our results show a significant reduction of the required computation time and memory compared to state estimation of TA using τ -DFA [10].

The remainder of this paper is structured as follows. Section 2 reviews related works. Section 3 provides relevant definitions and notation. Section 4 introduces our TE model and Sect. 5 provides an efficient method to derive TE from TA. In Sect. 6, we illustrate how TE can be used directly for verification, using the verification of current-state opacity as an example. In Sect. 7, we evaluate our method on 11 realistic case studies, and Sect. 8 concludes this paper.

2 Related Work

State estimation is a fundamental method in control theory. An estimator is a deterministic representation of a target system that provides the set of all possibly active states for any given input [25, 26]. In the literature, there exist multiple names for such a construction, like forward state estimator [26], current-state estimator [21], or observer [16, 28, 29]. For simplicity, we only use the term *estimator* for the remainder of this paper. In the following, we review related works on estimators for *timed automata* (TA).

The determinization, and therefore state estimation, of TA in the dense-time setting is generally not possible [6]. This means that there exist TA without any equivalent deterministic representation, and even checking if one exists is undecidable [6]. Related works on the dense-time setting therefore mainly identify decidable subclasses [4, 6, 7, 18]. These methods rely on the enumeration of single clock regions. This is equivalent to time step enumeration in a discrete-time setting, limiting the scalability for large systems.

Besides works concerning the dense-time setting, state estimation has also received attention in a discrete-time setting. There exist multiple works on estimator models of *maxplus automata*, *weighted automata*, or *automata over monoids* [13, 14, 16, 27]. In these automata models, transitions have attached weights that can be interpreted as time instances that enable the respective transitions. Such automata classes can therefore be seen as subclasses of TA in a discrete-time setting. However, as each transition resembles a single time instance, large automata are required to model realistic systems, heavily limiting scalability.

Another approach to compute estimators of discrete-time models is to use *tick automata* (τ -FA) [9]. Using a time abstraction, equivalent non-deterministic τ -FA can be derived, which can then be determinized using standard methods for finite automata [10, 19]. The result are *deterministic tick automata* (τ -DFA) which are estimators of the target system. However, τ -DFA model time using a special τ -symbol and therefore have to enumerate each time step individually. The scalability of this approach is thus also limited for realistic systems with large time constants. Figure 1 depicts our method in comparison. Our TE model is equally expressive as τ -DFA but more compact and efficient to compute. In Sect. 7, we compare both methods to compute estimators of a given TA.

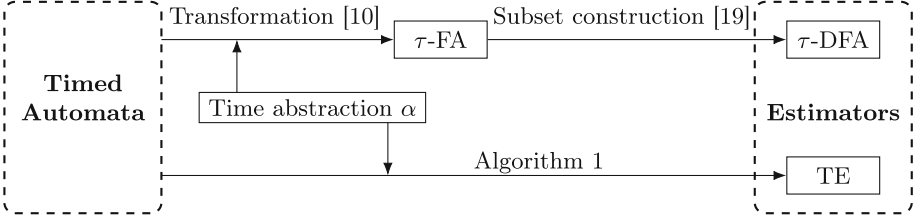


Fig. 1. Overview of our presented method compared to the approach in [10].

3 Preliminaries

In this section, we provide definitions and notation that we use throughout this paper. We first provide a formal definition of *timed automata* (TA) [3] and their *current-state estimates* (CSEs) in Sect. 3.1 to formally define our system model. In a second step, we introduce *deterministic tick automata* (τ -DFA) [9] and time abstractions in Sect. 3.2. τ -DFA are finite automata that can be used as estimators of TA under a time abstraction [10]. We use τ -DFA to illustrate and motivate key concepts of our new *threshold estimator* (TE) model in Sect. 4, and also as a base line in our evaluation in Sect. 7.

For general notation, we denote the empty word by ε , a sequence of two symbols σ, σ' by $\sigma\sigma'$, and a sequence of $k \in \mathbb{N}$ symbols σ by σ^k . For a given set S , let $\mathcal{P}(S)$ be the power set of S and $|S|$ the number of elements in S . An interval $I = [a, b]$ represents the set $\{i \in \mathbb{N} \mid a \leq i \leq b\}$, and let $L(I) := a$, and $R(I) := b$, for $a, b \in \mathbb{N}$ with $a \leq b$. The set of all such intervals is denoted by \mathcal{I} . Finally, we use the common assumption that $\sum_{i=j}^n i = 0$ holds if $n < j$.

3.1 Timed Automata and Current-State Estimates

We consider general TA in a discrete-time setting. We assume without loss of generality that a given TA only has a single clock, as TA can always be transformed to equivalent TA with a single clock in a discrete-time setting [8].

Definition 1 (Timed Automata). A *timed automaton* (TA) is a 5-tuple $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$, where

- L is a finite set of locations,
- $L_0 \subseteq L$ is a finite set of initial locations,
- Σ is a finite alphabet,
- $\Sigma_o \subseteq \Sigma$ is a finite set of observable symbols,
- $\Delta \subseteq L \times \Sigma \times \Gamma \times \{\top, \perp\} \times L$ is a finite set of transitions.

\mathcal{A} has a single clock c . The set of all possible guards Γ over c is defined by the grammar $\gamma := c = k \mid c \leq k \mid c \geq k \mid \gamma_1 \wedge \gamma_2$ with $k \in \mathbb{N}$, $\gamma_1, \gamma_2 \in \Gamma$. If c satisfies γ , we write $c \models \gamma$. A clock modifier $\lambda \in \{\top, \perp\}$ may reset c . This is formalized by time modification functions $f_\lambda : \mathbb{N} \rightarrow \mathbb{N}$, with $f_\top(n) = 0$ and $f_\perp(n) = n$.

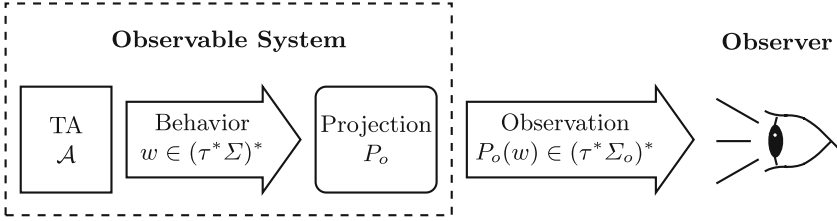


Fig. 2. The partially observable system model [10].

Note that states and locations are distinguished in TA. A state $\langle \ell, t \rangle$ is a location $\ell \in L$ at a specific time $t \in \mathbb{N}$. TA model timed behavior. As we use a discrete-time model, time flow can be modelled by sequences of discrete ticks. For this purpose, we use τ -steps. Given an alphabet Σ , a timed word w is therefore a sequence $w = \tau^{T_1} \sigma_1 \tau^{T_2} \sigma_2 \dots \tau^{T_n} \sigma_n \in (\tau^* \Sigma)^*$. We generally assume $\tau \notin \Sigma$ holds.

We consider TA in a partially observable setting. This means that an observer of the system can only observe the observable symbols $\sigma \in \Sigma_o$. The inferred projection function $P_o : \Sigma \rightarrow \Sigma_o$ is defined by $P_o(\sigma) = \sigma$ if $\sigma \in \Sigma_o$, and $P_o(\sigma) = \varepsilon$ if $\sigma \in \Sigma \setminus \Sigma_o$. P_o is extended to words with $P_o : (\tau^* \Sigma)^* \rightarrow (\tau^* \Sigma_o)^*$,

$$P_o(\tau^{T_1} \sigma_1 \tau^{T_2} \sigma_2 \dots \tau^{T_n} \sigma_n) = \tau^{T_1} P_o(\sigma_1) \tau^{T_2} P_o(\sigma_2) \dots \tau^{T_n} P_o(\sigma_n).$$

To model the view of an observer of a system, we use current-state estimates (CSEs) [29], which we adapt to TA as follows.

Definition 2 (Current-State Estimate). *Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$, a current-state estimate (CSE) $e \in \mathcal{P}(L \times \mathbb{N})$ is a set of states that could currently be active with*

$$e = \{ \langle \ell_1, t_1 \rangle, \langle \ell_2, t_2 \rangle, \dots, \langle \ell_n, t_n \rangle \} \in \mathcal{P}(L \times \mathbb{N}).$$

Let $\varepsilon\text{-CLO} : \mathcal{P}(L \times \mathbb{N}) \rightarrow \mathcal{P}(L \times \mathbb{N})$ be the standard ε -closure [19] with

$$\varepsilon\text{-CLO}(e) = e \cup \varepsilon\text{-CLO}(\{ \langle \ell', f_\lambda(t) \rangle \mid \langle \ell, t \rangle \in e \wedge (\ell, \sigma, \gamma, \lambda, \ell') \in \Delta \wedge P_o(\sigma) = \varepsilon \wedge t \models \gamma \}),$$

and for $k \in \mathbb{N}$, let $e \oplus k := \{ \langle \ell, t + k \rangle \mid \langle \ell, t \rangle \in e \}$ be the the modified addition for CSEs and natural numbers.

Intuitively, a CSE can be seen as an estimate from an outside observer that tracks currently active states. Consider Fig. 2. We assume that an observer has only access to the observable symbols $\sigma \in \Sigma_o$. The original behavior of a TA is therefore projected by P_o . This makes TA generally non-deterministic from the view of an observer. A CSE can therefore contain multiple states. $\varepsilon\text{-CLO}$ is the standard ε -closure, adapted to CSEs [19] and provides all states that could be reached by unobservable transitions. $\varepsilon\text{-CLO}$ is therefore an essential tool for state estimation.

Example 1. An example TA $\mathcal{A}_E = (L, L_0, \Sigma, \Sigma_o, \Delta)$ is depicted in Fig. 3a with $\Sigma = \{o, u\}$ and $\Sigma_o = \{o\}$, and the locations $L = \{\ell_1, \ell_2, \ell_3\}$. ℓ_1 is the only initial location with $L_0 = \{\ell_1\}$. Each transition $(\ell, \sigma, \gamma, \lambda, \ell') \in \Delta$ is depicted by an arc with label σ, γ, λ . A CSE of \mathcal{A}_E could be $\{\langle \ell_2, 4 \rangle, \langle \ell_3, 4 \rangle\}$. This would mean that either $\langle \ell_2, 4 \rangle$ or $\langle \ell_3, 4 \rangle$ is currently active.

For a given TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$, let $\delta_{\mathcal{A}} : \mathcal{P}(L \times \mathbb{N}) \times (\Sigma_o \cup \{\tau\}) \rightarrow \mathcal{P}(L \times \mathbb{N})$ be the inferred observable transition function from Δ with

$$\begin{aligned} \delta_{\mathcal{A}}(e, \tau) &= \varepsilon\text{-CLO}(e \oplus 1), \\ \delta_{\mathcal{A}}(e, \sigma) &= \varepsilon\text{-CLO}(\{\langle \ell', f_{\lambda}(t) \rangle \mid \langle \ell, t \rangle \in e \wedge (\ell, \sigma, \gamma, \lambda, \ell') \in \Delta \wedge t \models \gamma\}). \end{aligned}$$

$\delta_{\mathcal{A}}$ is extended recursively with $\delta_{\mathcal{A}} : \mathcal{P}(L \times \mathbb{N}) \times (\Sigma_o \cup \{\tau\})^* \rightarrow \mathcal{P}(L \times \mathbb{N})$ and $\delta_{\mathcal{A}}(e, ua) = \bigcup_{e' \in \delta_{\mathcal{A}}(e, u)} \delta_{\mathcal{A}}(e', a)$, for $u \in (\Sigma_o \cup \{\tau\})^*$, $a \in (\Sigma_o \cup \{\tau\})^*$, and $e \in \mathcal{P}(L \times \mathbb{N})$. $\delta_{\mathcal{A}}$ therefore characterizes every reachable CSE of \mathcal{A} .

3.2 Time Abstractions and Tick Automata

TA in a discrete-time setting can be transformed to τ -DFA [9]. τ -DFA are standard FA that use the τ -symbol to model the passing of one time unit. The key to this transformation is the use of a time abstraction α . TA generally have an infinite state space, as each location can be active for any clock valuation. The infinite number of possible clock values then leads to an infinite state space. A time abstraction can overcome this problem by grouping clock values in finitely many equivalence classes [9, 10]. The key idea is that all clock values of the same equivalence class lead to equivalent behavior in the TA. The finite set of equivalence classes can then be seen as the state space for a τ -DFA. In [9], the region abstraction α_R [3] is used to transform TA to equivalent τ -DFA. In [10], a local time abstraction α_L is introduced that can be used for a similar but more efficient transformation. We adapt both notions to our model as follows.

Definition 3 (Time Abstraction). *Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$, the region abstraction $\alpha_R : \mathbb{N} \rightarrow \mathbb{N}$ and the local time abstraction $\alpha_L : L \times \mathbb{N} \rightarrow \mathbb{N}$ are defined as follows*

$$\alpha_R(v) = \begin{cases} M_{\mathcal{A}} + 1, & \text{if } v > M_{\mathcal{A}}, \\ v, & \text{otherwise,} \end{cases} \quad \alpha_L(\ell, v) = \begin{cases} M_{\mathcal{A}}^{\ell} + 1, & \text{if } v > M_{\mathcal{A}}^{\ell}, \\ v, & \text{otherwise.} \end{cases}$$

such that $M_{\mathcal{A}}$ is the largest constant in any guard in \mathcal{A} , and $M_{\mathcal{A}}^{\ell}$ is the largest local constant of ℓ in \mathcal{A} . For detail on how to compute $M_{\mathcal{A}}^{\ell}$, we refer to [10]. We extend both α_R and α_L to CSEs with

$$\begin{aligned} \alpha_R : \mathcal{P}(L \times \mathbb{N}) &\rightarrow \mathcal{P}(L \times \mathbb{N}), & \alpha_L : \mathcal{P}(L \times \mathbb{N}) &\rightarrow \mathcal{P}(L \times \mathbb{N}), \\ \alpha_R(e) &= \{\langle \ell, \alpha_R(t) \rangle \mid \langle \ell, t \rangle \in e\}, & \alpha_L(e) &= \{\langle \ell, \alpha_L(\ell, t) \rangle \mid \langle \ell, t \rangle \in e\}. \end{aligned}$$

Intuitively, a time abstraction α reduces the infinite set $\mathcal{P}(L \times \mathbb{N})$ of possible CSEs to a finite set $E = \{\alpha(e) \mid e \in \mathcal{P}(L \times \mathbb{N})\}$. We can use this finite set E as the state set of a τ -DFA as follows.

Definition 4 (Deterministic Tick Automata). Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$ and a time abstraction α , a deterministic tick automaton (τ -DFA) is a 4-tuple $A = (E, e_0, \Sigma_o \cup \{\tau\}, \delta_A)$, where

- $E = \{\alpha(e) \mid e \in \mathcal{P}(L \times \mathbb{N})\} \subset \mathcal{P}(L \times \mathbb{N})$ is a finite set of states,
- $e_0 = \{\langle \ell_0, 0 \rangle \mid \ell_0 \in L_0\}$ is the unique initial state,
- $\delta_A : E \times (\Sigma_o \cup \{\tau\}) \rightarrow E$ is a transition function, with

$$\begin{aligned} \delta_A(e, \tau) &= \alpha(e \oplus 1), \\ \delta_A(e, \sigma) &= \alpha(\{\langle \ell', f_\lambda(t) \rangle \mid \langle \ell, t \rangle \in e \wedge (\ell, \sigma, \gamma, \lambda, \ell') \in \Delta \wedge t \models \gamma\}), \end{aligned}$$

and extended to timed words with $\delta_A : E \times (\Sigma_o \cup \{\tau\})^* \rightarrow E$, and $\delta_A(e, ua) = \bigcup_{e' \in \delta_A(e, u)} \delta_A(e', a)$, for $u \in (\Sigma_o \cup \{\tau\})^*$, $a \in (\Sigma_o \cup \{\tau\})$, and $e \in \mathcal{P}(L \times \mathbb{N})$.

Intuitively, τ -DFA model the behavior of a TA \mathcal{A} under a time abstraction α . τ -DFA use CSEs as a state set E . To model the time flow, each CSE $e \in E$ has a τ -transition to its successor CSE $\alpha(e \oplus 1)$. A transition $e \xrightarrow{\sigma} e'$ exists if the CSE e would lead to e' for all enabled σ -transitions in e . The behavior of \mathcal{A} and a derived τ -DFA are therefore equivalent.

To compute τ -DFA, TA are first transformed to equivalent τ -FA [10]. The resulting τ -FA are not necessarily deterministic. Using standard determinization methods for FA [19], we can then compute equivalent τ -DFA. We depict this method in Fig. 1. For details on how to compute a τ -DFA, we refer to [10].

Example 2. Consider our example TA \mathcal{A}_E (Fig. 3a). An equivalent τ -DFA A_E is depicted in Fig. 3b. α_R is used to compute the state space. A_E enumerates all distinguishable clock valuations under the time abstraction α_R . Intuitively, each clock value $c \geq 5$ cannot be distinguished from $c = 5$. It is therefore sufficient to only consider clock values $c \leq 5$. Each state in A_E that can be reached by a word w is an accurate CSE of \mathcal{A}_E .

3.3 Current-State Opacity

One application of a state estimation procedure is to verify privacy properties of a given system [28]. *Current-state opacity* (CSO) is a privacy property that can be directly verified using estimators [10]. In the CSO setting, the set of possible system states is divided in secret and non-secret states. An intruder has full knowledge of the systems structure and all observable events. CSO holds if an intruder cannot deduce that a secret state is currently active. We adapt the notion of CSO to our setting as follows.

Definition 5 (Current-State Opacity). Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$, and a set of secret locations $L_S \subseteq L$, current-state opacity (CSO) holds in \mathcal{A} if $\delta_{\mathcal{A}}(\{\langle \ell_0, 0 \rangle \mid \ell_0 \in L_0\}, w) \setminus (L_S \times \mathbb{N}) \neq \emptyset$ holds for any word $w \in (\tau^* \Sigma_o)^*$.

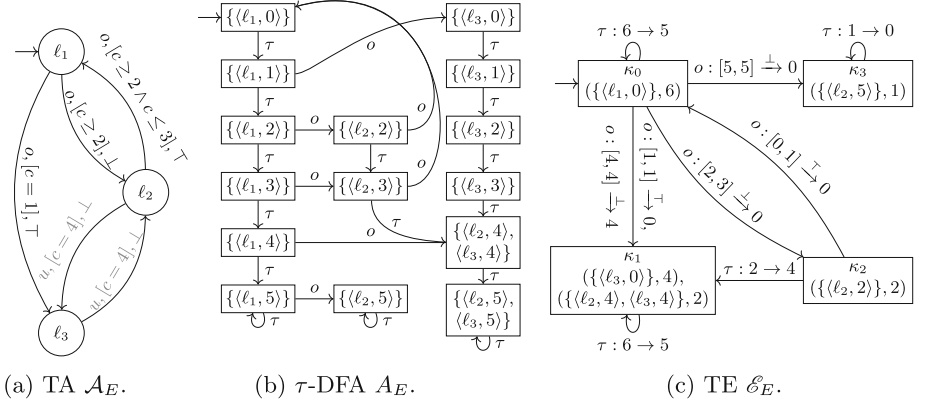


Fig. 3. Running example with equivalent estimators assuming u is not observable.

Intuitively, Definition 5 states that every reachable CSE in \mathcal{A} contains at least one non-secret state. Or in other words, for every word w that leads to a secret state, there also exists a word w' that leads to a non-secret state and cannot be distinguished from w . We provide an example for the application of our method to verify CSO of TA in Sect. 6.

4 Threshold Estimation

In this section, we introduce our *threshold estimator* (TE), a new estimator model for *timed automata* (TA). The key idea for the TE model is to avoid the evaluation of every single time instance. Instead, we identify and evaluate only significant time instances (thresholds) that are necessary for the computation of an estimator. We thereby aim to reduce the computational costs for TE and enable efficient verification of TA as we discuss later in Sect. 6.

In the following, we first introduce our time flow estimation as a base for the TE model in Sect. 4.1. We then provide a formal definition for our TE model in Sect. 4.2 and prove that it can be derived from any TA.

4.1 Estimating Time Flows

One key problem of state estimation in a timed system is to estimate its time flow. This means to determine the future of a given state with regards to time flow. However, this is not trivial, as a state in a TA can have multiple possible future states via unobservable transitions. The usual way to resolve unobservable transitions in untimed systems is to merge states using the standard ε -closure [19]. A similar approach can be used in a timed setting. However, the ε -closure would then be required at every time step.

Example 3. Consider the TA \mathcal{A}_E (Fig. 3a) and the CSE $\{\langle \ell_2, 2 \rangle\}$. To compute the future of $\{\langle \ell_2, 2 \rangle\}$, we can iteratively advance the CSE by 1 and get the infinite sequence $\{\langle \ell_2, 2 \rangle\} \xrightarrow{\tau} \{\langle \ell_2, 3 \rangle\} \xrightarrow{\tau} \dots$. However, note that in $\{\langle \ell_2, 4 \rangle\}$, the unobservable transition to ℓ_3 is enabled. $\langle \ell_3, 4 \rangle$ could therefore also be active from the point of view of an observer. This means $\langle \ell_2, 3 \rangle$ has multiple possible future states. In order to accurately compute the future of a CSE e , we therefore have to apply ε -CLO at every step and get $\varepsilon\text{-CLO}(e) \xrightarrow{\tau} \varepsilon\text{-CLO}(\varepsilon\text{-CLO}(e) \oplus 1) \xrightarrow{\tau} \dots$. For $\{\langle \ell_2, 2 \rangle\}$, we thereby get $\{\langle \ell_2, 2 \rangle\} \xrightarrow{\tau} \{\langle \ell_2, 3 \rangle\} \xrightarrow{\tau} \{\langle \ell_2, 4 \rangle, \langle \ell_3, 4 \rangle\} \xrightarrow{\tau} \dots$. Notice that ε -CLO only has an impact for $\{\langle \ell_2, 4 \rangle\}$. At any other point, the successor of a CSE e , is just $e \oplus 1$ in this example.

The key idea of our estimation approach for a CSE e is to determine the next threshold $s \in \mathbb{N}^+$, at which new states can be reached, by unobservable-transitions. All smaller time steps $s' < s$ do not have to be evaluated and characterize the inferred CSEs with $e \oplus s'$. By using a time abstraction, we only need to consider a finite number of thresholds s . We formalize this idea as follows.

Definition 6 (Timed ε -Closure). *Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$ and a time abstraction α , let $\tau\text{-}\varepsilon\text{-CLO} : \mathcal{P}(L \times \mathbb{N}) \rightarrow (\mathcal{P}(L \times \mathbb{N}) \times \mathbb{N})^+$ be a function with $\tau\text{-}\varepsilon\text{-CLO}(e) = [(e_1, s_1), \dots, (e_n, s_n)]$, where $e_1 := e$ and for all $i \in [1, n-1]$, let $e_{i+1} = \alpha(\varepsilon\text{-CLO}(e_i \oplus s_i))$ where $s_i \in \mathbb{N}^+$ is the smallest value such that*

$$\alpha(\varepsilon\text{-CLO}(e_i \oplus s_i)) \neq \alpha(\varepsilon\text{-CLO}(e_i \oplus (s_i - 1))). \quad (1)$$

Further, $s_n \in \mathbb{N}$ is the smallest possible value such that there exist $j \in [1, n]$ and $k \in \mathbb{N}$ with $k < s_j$ such that $\alpha(e_n \oplus s_n) = \alpha(e_j \oplus k)$.

Intuitively, $\tau\text{-}\varepsilon\text{-CLO}(e)$ represents the estimated time flow of a CSE e . s_i is the threshold that needs to be evaluated for the current e_i for $1 \leq i < n$. For all smaller time steps $s < s_i$, the CSEs can be inferred by $e_i \oplus s$. This is a significant advantage over the computation of τ -DFA, where every state must be evaluated. By the use of a time abstraction α , we guarantee that we always find a previously visited CSE at some point, as α limits all possible CSEs to a finite set (see Definition 3). Since each s_i is the smallest value in a finite time space, we guarantee its existence and unicity. To compute each s_i , it is sufficient to check Inequality 1 only for time instances that enable unobservable transitions that lead to new states. Note that in the worst case, we still get $s_i = 1$ for all $i \in [1, n]$, meaning we have to evaluate every single time instance like in τ -DFA. The worst case complexity is therefore the same. Having defined $\tau\text{-}\varepsilon\text{-CLO}$, we have the following result.

Lemma 1. *Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$, a time abstraction α , and a CSE $e \in \mathcal{P}(L \times \mathbb{N})$, with $\tau\text{-}\varepsilon\text{-CLO}(e) = [(e_1, s_1), (e_2, s_2), \dots, (e_n, s_n)]$, for any $z \in \mathbb{N}$, let $j := \max_{j \in [1, n]} \sum_{i=1}^{j-1} s_i \leq z$. Then, $\alpha(\delta_{\mathcal{A}}(e, \tau^z)) = \alpha(e_j \oplus s)$ with $s = z - \sum_{i=1}^{j-1} s_i$ holds.*

Proof. Let $e \in \mathcal{P}(L \times \mathbb{N})$ and let $\tau\text{-}\varepsilon\text{-CLO}(e) = [(e_1, s_1), (e_2, s_2), \dots, (e_n, s_n)]$.
(A): By Def. 6, for $j < n$, we have $\alpha(\delta_{\mathcal{A}}(e, \tau^{\hat{z}})) = \alpha(e_{j+1})$ with $\hat{z} = \sum_{i=1}^j s_i$.

(B): Assume for $z \in \mathbb{N}$ there exists $j \in [1, n]$ with $\alpha(\delta_{\mathcal{A}}(e, \tau^z)) = \alpha(e_j \oplus s)$ and $s = z - \sum_{i=1}^{j-1} s_i$. This implies that for all $k \in [0, s_j - 1]$ we have $\alpha(\delta_{\mathcal{A}}(e, \tau^{z+k})) = \alpha(e_j \oplus s + k)$ by Def. 6.

Note that due to **(A)**, Lemma 1 holds for each $z \in Z := \{\sum_{i=1}^j s_i \mid j \in [1, n-1]\}$. Such z satisfy **(B)**, meaning that also for all $z' \in ([0, \sum_{i=1}^{n-1} s_i] \setminus Z)$. Lemma 1 holds. Finally, we have $\tilde{z} = \sum_{i=1}^n s_i \wedge \alpha(\delta_{\mathcal{A}}(e, \tau^{\tilde{z}})) = \alpha(e_n \oplus s_n)$ and **(B)**, and therefore, Lemma 1 holds for all $z'' > \tilde{z}$.

Intuitively, Lemma 1 states that τ - ε -CLO(e) can be used to infer any future CSE of a given CSE e under a time abstraction α .

Example 4. Consider the TA \mathcal{A}_E (Fig. 3a) under the region abstraction α_R . Using τ - ε -CLO, we can accurately estimate the time flow of any CSE. Consider τ - ε -CLO($\{\langle \ell_1, 0 \rangle\}$) = $[[\{\langle \ell_1, 0 \rangle\}, s_1 := 6]]$, and τ - ε -CLO($\{\langle \ell_2, 2 \rangle\}$) = $[[\{\langle \ell_2, 2 \rangle\}, s'_1 := 2], \{\langle \ell_2, 4 \rangle, \langle \ell_3, 4 \rangle\}, s'_2 := 2]]$. Note that for $\langle \ell_1, 0 \rangle$, there exist no unobservable transitions that can lead to future states. We therefore only have a single element in τ - ε -CLO($\{\langle \ell_1, 0 \rangle\}$). Moreover, $\alpha_R(\{\langle \ell_1, k \rangle\}) = \{\langle \ell_1, 5 \rangle\}$ for all $k \geq 6$ holds. The first step at which a previously visited state is encountered is therefore $s_1 = 6$. As discussed in Example 3, $\langle \ell_2, 4 \rangle$ enables the unobservable transition to ℓ_3 . In τ - ε -CLO($\{\langle \ell_2, 2 \rangle\}$), we therefore have two elements. Starting in $\{\langle \ell_2, 2 \rangle\}$, we encounter $\{\langle \ell_2, 4 \rangle, \langle \ell_3, 4 \rangle\}$ after $s'_1 = 2$ steps. We also have $\alpha_R(\{\langle \ell_2, k \rangle, \langle \ell_3, k \rangle\}) = \{\langle \ell_2, 5 \rangle, \langle \ell_3, 5 \rangle\}$ for all $k \geq 6$. So from $\{\langle \ell_2, 4 \rangle, \langle \ell_3, 4 \rangle\}$, we encounter a previously visited state after $s'_2 = 2$ steps. Consider the τ -DFA A_E of \mathcal{A}_E (Fig. 3a), depicted in Fig. 3b. A_E contains the states $\{\langle \ell_1, 0 \rangle\}$ and $\{\langle \ell_2, 2 \rangle\}$. Each future CSE can be reached via τ -transitions. All such future states can also be inferred by our more compact representation of τ - ε -CLO.

4.2 Threshold Estimators

Definition 7 (Threshold Estimator). *Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$, a threshold estimator (TE) is a 6-tuple $\mathcal{E} = (K, \kappa_0, \xi, \Sigma_o, \delta_\tau, \Delta_{\mathcal{E}})$, where*

- $K \subseteq \mathcal{P}(L \times \mathbb{N})$ is a finite set of locations,
- $\xi : K \times \mathbb{N} \rightarrow \mathcal{P}(L \times \mathbb{N})$ is the estimation function,
- $\kappa_0 = \{\langle \ell_0, 0 \rangle \mid \ell_0 \in L_0\}$ is the initial location,
- $\delta_\tau : K \times \mathbb{N} \rightarrow K \times \mathbb{N}$ is the delay transition function,
- $\Delta_{\mathcal{E}} \subseteq K \times \sigma \times \mathcal{I} \times \{\top, \perp\} \times \mathbb{N} \times K$ is the finite set of interval transitions.

Intuitively, each location $\kappa \in K$ in \mathcal{E} models a CSE in \mathcal{A} . Let τ - ε -CLO(κ) = $[(e_1, s_1), (e_2, s_2), \dots, (e_n, s_n)]$ and let $\eta_\kappa = \sum_{i=1}^n s_i$. Using τ - ε -CLO(κ) we characterize the estimated time flow in κ until η_κ as follows. We define $\xi(\langle \kappa, t \rangle) := \alpha(e_j \oplus s)$, with the largest possible $j \in [1, n]$, such that $\sum_{i=0}^{j-1} s_i \leq t < \eta_\kappa$, and $s := t - \sum_{i=0}^{j-1} s_i$. Intuitively, ξ provides the future CSEs of κ at any point in time. So for a state $\langle \kappa, t \rangle \in K \times \mathbb{N}$ in \mathcal{E} , where κ is a CSE in \mathcal{A} , $\xi(\langle \kappa, t \rangle)$ is the CSE in \mathcal{A} , reached from κ after t time steps.

Further, let $\delta_\tau(\kappa, \eta_\kappa) := \langle \kappa', t' \rangle$ and for all other $t \in \mathbb{N}$ with $t \neq \eta_\kappa$, let $\delta_\tau(\kappa, t) = \langle \kappa, t \rangle$. Intuitively, η_κ limits the time that can be spent in κ and with $\delta_\tau(\kappa, \eta_\kappa)$ we define the time flow for the next step to a new state $\langle \kappa', t' \rangle$.

We assume that for any $\kappa \in K, \sigma \in \Sigma$, and $t \in \mathbb{N}$, there exists exactly one $(\kappa, \sigma, I, \lambda, z, \kappa') \in \Delta_\mathcal{E}$ such that $t \in I$ and $R(I) < \eta_\kappa$ always holds. This ensures that \mathcal{E} is deterministic. The initial state of \mathcal{E} is $\langle \kappa_0, 0 \rangle$ and the behavior of \mathcal{E} is defined by $\delta_\mathcal{E} : K \times \mathbb{N} \times (\Sigma_o \cup \{\tau\}) \rightarrow K \times \mathbb{N}$ as follows.

$$\begin{aligned} \delta_\mathcal{E}(\langle \kappa, t \rangle, \tau) &= \delta_\tau(\langle \kappa, t + 1 \rangle), \\ \delta_\mathcal{E}(\langle \kappa, t \rangle, \sigma) &= \langle \kappa', f_\lambda(t, z) \rangle \text{ such that } \exists!(\kappa, \sigma, I, \lambda, z, \kappa') \in \Delta_\mathcal{E} \wedge t \in I, \end{aligned}$$

where $f_\lambda(t, z) := z$ if $\lambda = \top$, and $f_\lambda(t, z) := z + t - L(I)$ if $\lambda = \perp$.

We extend $\delta_\mathcal{E}$ to words with $\delta_\mathcal{E} : K \times \mathbb{N} \times (\Sigma_o \cup \{\tau\})^* \rightarrow K \times \mathbb{N}$ and $\delta_\mathcal{E}(s, ua) = \bigcup_{s' \in \delta_\mathcal{E}(s, u)} \delta_\mathcal{E}(s', a)$, for $u \in (\Sigma_o \cup \{\tau\})^*$, $a \in (\Sigma_o \cup \{\tau\})^*$, and $s \in K \times \mathbb{N}$.

Example 5. Consider the TA \mathcal{A}_E (Fig. 3a). A TE \mathcal{E}_E of \mathcal{A}_E is depicted in Fig. 3c. \mathcal{E}_E is computed using Algorithm 1, which we present later in Sect. 5. We depict each delay transition for $\delta_\tau(\kappa, \eta_\kappa) = \langle \kappa', t \rangle$ by an arc from κ to κ' , labeled $\tau : \eta_\kappa \rightarrow t$. Furthermore, we depict each transition $(\kappa, \sigma, I, \lambda, t, \kappa') \in \Delta_\mathcal{E}$ by an arc from κ to κ' , labeled $\sigma : I \xrightarrow{\lambda} t$. Notice that κ_1 and κ_2 share the future CSE $\{\langle \ell_2, 4 \rangle, \langle \ell_3, 4 \rangle\}$. We therefore have a delay transition from κ_2 to κ_1 . Note that \mathcal{E}_E represents the same deterministic behavior as the τ -DFA A_E (Fig. 3b) in a more compact way.

Intuitively, TE are slightly modified TA with a single clock that can be set to specific values. Moreover, TE have a maximal delay η_κ for each location κ to explicitly model time flow $\delta_\tau(\langle \kappa, \tau \rangle)$ leading to different locations, like τ -DFA. Having defined TE, we have the following result.

Theorem 1. *Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$ and a time abstraction α , there always exists a TE $\mathcal{E} = (K, \kappa_0, \xi, \Sigma_o, \delta_\tau, \Delta_\mathcal{E})$, such that for any $w \in (\tau^* \Sigma_o)^*$ the following holds*

$$\alpha(\delta_\mathcal{A}(\{\langle l_0, 0 \rangle \mid l_0 \in L_0\}, w)) = \xi(\delta_\mathcal{E}(\langle \kappa_0, 0 \rangle, w)).$$

Proof. Let $e \in \mathcal{P}(L \times \mathbb{N})$, and let τ - ε -CLO(e) = $[(e_1, s_1), (e_2, s_2), \dots, (e_n, s_n)]$. By Lemma 1, for any $T \in \mathbb{N}$, there exists a largest $j \in [1, n]$, with $\alpha(\delta_\mathcal{A}(e, \tau^T)) = \alpha(e_j \oplus s)$ with $s := T - \sum_{i=1}^{j-1} s_i$ and $\sum_{i=1}^{j-1} s_i \leq T$. Let $\kappa := e$ and assume $T \geq \eta_\kappa$. By Definition 7, we have $\delta_\mathcal{E}(\kappa, \eta_\kappa) = \langle \kappa^1, t^1 \rangle$ with $\xi(\langle \kappa, \eta_\kappa \rangle) = \xi(\langle \kappa^1, t^1 \rangle)$. Now, let $T^1 := T - \eta_\kappa$. If $T^1 \geq \eta_{\kappa^1}$, we again have $\delta_\mathcal{E}(\kappa^1, \eta_{\kappa^1}) = \langle \kappa^2, t^2 \rangle$ with $\xi(\langle \kappa^1, \eta_{\kappa^1} \rangle) = \xi(\langle \kappa^2, t^2 \rangle)$. We repeat this process until we reach $\langle \kappa^m, t^m \rangle$ with $T^m < \eta_{\kappa^m}$ for some $m \in \mathbb{N}$. Then by Definition 7, we have $\xi(\langle \kappa^m, T^m \rangle) = \delta_\mathcal{E}(\langle \kappa, 0 \rangle, \tau^T) = \alpha(e_j \oplus s) = \alpha(\delta_\mathcal{A}(e, \tau^T))$. Now, let $\hat{e} \in \mathcal{P}(L \times \mathbb{N})$, and let $\hat{\kappa} \in K, \hat{t} < \eta_{\hat{\kappa}}$, with $\xi(\langle \hat{\kappa}, \hat{t} \rangle) = e$. For any $\hat{t} \in [\hat{t}, \eta_{\hat{\kappa}}]$, we add a transition

$$(\hat{\kappa}, \sigma, [\hat{t}, \hat{t}], \top, 0, \hat{\kappa}'), \quad (2)$$

such that $\hat{\kappa}' := \{\langle \ell', f_\lambda(t) \rangle \mid \langle \ell, t \rangle \in \hat{e} \wedge (\ell, \sigma, \gamma, \lambda, \ell') \wedge t + \tilde{t} \models \gamma\}$. We therefore have $\delta_{\mathcal{E}}(\langle \kappa, 0 \rangle, \tau^T \sigma) = \alpha(\delta_{\mathcal{A}}(e, \tau^T \sigma))$. Generally, for $w \in (\tau^* \Sigma_o)^*$, we therefore have $\alpha(\delta_{\mathcal{A}}(\{\langle l_0, 0 \rangle \mid l_0 \in L_0\}, w)) = \xi(\delta_{\mathcal{E}}(\langle \kappa_0, 0 \rangle, w))$.

With Theorem 1, we have proven that for any TA, there also exists a TE that provides an accurate CSE at any point in time. The construction we have used for the observable transitions is equivalent to the transitions in τ -DFA. Intuitively, we therefore also can derive an equivalent τ -DFA from any TE. With Theorem 1, this implies that TE and τ -DFA are equally expressive, since TA and τ -DFA are also equally expressive [10].

The construction in the above proof is simple and intuitive but inefficient. In the next section we show how to construct more compact TE more efficiently with significantly fewer transitions.

5 Efficient Threshold Estimator Synthesis

In this section, we present our method to efficiently derive *threshold estimators* (TE) from *timed automata* (TA). In principle, we could use the method in the proof of Theorem 1 to derive equivalent TE from TA. However, such TE would have transitions for every single time instance, leading to large models like τ -DFA. We can compute significantly more compact TE by grouping multiple time instances into single transitions with longer intervals.

Example 6. Consider ℓ_2 in \mathcal{A}_E (Fig. 3a). ℓ_2 has a transition for o to ℓ_1 that resets the clock c . Assume we have $\kappa := \{\langle \ell_2, 2 \rangle\}$. Using the method in the proof of Theorem 1, we would get the transitions

$$(\kappa, o, [0, 0], \top, 0, \{\langle \ell_1, 0 \rangle\}) \text{ and } (\kappa, o, [1, 1], \top, 0, \{\langle \ell_1, 0 \rangle\}).$$

Both transitions together are equivalent to the single transition

$$(\kappa, o, [0, 1], \top, 0, \{\langle \ell_1, 0 \rangle\}).$$

Intuitively, as both transitions have the same target state, it would be sufficient to only evaluate a single time instance (0 or 1). We can thereby reduce the computational costs of computing such transitions in a TE.

In the following, we first identify cases where we can merge multiple time instances in a single transition in Sect. 5.1. Second, we utilize these results in Algorithm 1 to efficiently derive more compact TE from TA in Sect. 5.2.

5.1 Grouping Transitions

We now identify cases in which we can merge transitions in the construction of TE and reduce computational costs. Let $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$ be a TA and let $e \in \mathcal{P}(L \times \mathbb{N})$ be a CSE in \mathcal{A} . Recall that we can estimate the time flow of e by a location $\kappa := e$, as explained in Sect. 4.2. We illustrate this idea in Fig. 4. With

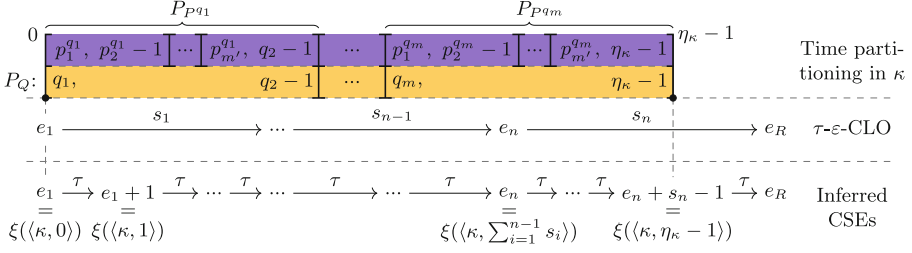


Fig. 4. Partitioning of observable behavior for κ in the interval ${}_{\kappa}I = [0, \eta_{\kappa} - 1]$.

τ - ε -CLO(e) = $[(e_1, s_1), \dots, (e_n, s_n)]$, depicted in the middle row, we can derive the inferred states using ξ , depicted in the lower row. At $\eta_{\kappa} = \sum_{i=1}^n s_i$ we reach a previously visited CSE $e_R \in \mathcal{P}(L \times \mathbb{N})$. By Definition 6, e_R must always exist. We therefore only need to consider the interval ${}_{\kappa}I := [0, \eta_{\kappa} - 1]$ to compute the observable transitions in κ . Let $f_{\Delta} : \mathcal{P}(L \times \mathbb{N}) \times \Sigma_o \rightarrow \mathcal{P}(\Delta)$ be a function to compute the set of all enabled observable transitions in e with

$$f_{\Delta}(e, \sigma) = \{(\ell, \sigma, \gamma, \lambda, \ell') \in \Delta \mid \langle \ell, t \rangle \in e \wedge t \models \gamma\}.$$

We now partition ${}_{\kappa}I$ such that we get sub intervals in which at any point in time, the set of enabled observable transitions remains the same. The purpose is to identify time spans in which we can group transitions to intervals. Fix $\sigma \in \Sigma_o$ and let $Q = \{q_1, q_2, \dots, q_m\}$ be a set of threshold values such that $q_1 := L({}_{\kappa}I) = 0$ and for all q_j with $j > 1$ we have $q_j \in {}_{\kappa}I \wedge q_j > q_{j-1}$, and

$$f_{\Delta}(\xi(\langle \kappa, q_j \rangle), \sigma) \neq f_{\Delta}(\xi(\langle \kappa, q_j - 1 \rangle), \sigma). \quad (3)$$

Q induces a partition P_Q of ${}_{\kappa}I$ with

$$P_Q = \{[L({}_{\kappa}I) = 0 = q_1, q_2 - 1], [q_2, q_3 - 1], \dots, [q_m, R({}_{\kappa}I)]\}.$$

For any ${}_{\kappa}^q I \in P_Q$ and any $a, b \in {}_{\kappa}^q I$ we have $f_{\Delta}(\xi(\langle \kappa, a \rangle), \sigma) = f_{\Delta}(\xi(\langle \kappa, b \rangle), \sigma)$.

In Fig. 4, P_Q is depicted at the top in yellow and can be computed by using the state information, inferred from ξ . Each threshold $q \in Q$ can be computed iteratively by testing only the border values of the guards of all reachable transitions from κ in Inequality 3. In the worst case, we get $\eta_{\kappa} - 1$ intervals, resulting in a worst case complexity of $O((\eta_{\kappa} - 1) \cdot |L| \cdot |\Delta|)$ to compute P_Q .

Note that for $a, b \in {}_{\kappa}^q I$ generally $\delta_{\mathcal{A}}(\xi(\langle \kappa, a \rangle), \sigma) \neq \delta_{\mathcal{A}}(\xi(\langle \kappa, b \rangle), \sigma)$ can hold. This makes grouping transitions in ${}_{\kappa}^q I$ not trivial. We now present special cases in which we can group transitions to intervals. Consider an interval ${}_{\kappa}^q I \in P_Q$.

Case 1: Assume for all $(\ell, \sigma, \gamma, \lambda, \ell') \in f_{\Delta}(\xi(\langle \kappa, L({}_{\kappa}^q I) \rangle), \sigma)$ we have $\lambda = \top$. In this case, we have $\delta_{\mathcal{A}}(\xi(\langle \kappa, a \rangle), \sigma) = \delta_{\mathcal{A}}(\xi(\langle \kappa, b \rangle), \sigma) = \{\langle \ell', 0 \rangle \mid (\ell, \sigma, \gamma, \lambda, \ell') \in f_{\Delta}(\xi(\langle \kappa, L({}_{\kappa}^q I) \rangle), \sigma)\}$ for any $a, b \in {}_{\kappa}^q I$. We can therefore represent the transitions of $f_{\Delta}(\xi(\langle \kappa, L({}_{\kappa}^q I) \rangle), \sigma)$ in ${}_{\kappa}^q I$ by the transition

$$(\kappa, \sigma, {}_{\kappa}^q I, \top, t, \kappa'), \text{ with } \xi(\langle \kappa', t \rangle) = \delta_{\mathcal{A}}(\xi(\langle \kappa, L({}_{\kappa}^q I) \rangle), \sigma). \quad (4)$$

Case 2: Assume for all $(\ell, \sigma, \gamma, \lambda, \ell') \in f_{\Delta}(\xi(\langle \kappa, L(\kappa, I) \rangle), \sigma)$ we have $\lambda = \perp$. Since the clock is not reset, we can group the transitions if the target states do not enable unobservable transitions. For this purpose, we partition ${}^q_{\kappa}I$ again as follows. Let $P^q = \{L({}^q_{\kappa}I) =: p_1^q, p_2^q, \dots, p_{m'}^q\}$ be a set of threshold values such that for all p_j^q , with $j > 1$, we have $p_j^q \in {}^q_{\kappa}I \wedge p_j^q > p_{j-1}^q$ with

$$\varepsilon\text{-CLO}(\delta_{\mathcal{A}}(\xi(\langle \kappa, p_j^q \rangle), \sigma)) \neq \varepsilon\text{-CLO}(\delta_{\mathcal{A}}(\xi(\langle \kappa, p_j^q - 1 \rangle), \sigma)) \oplus 1. \quad (5)$$

P^q induces a partition P_{P^q} of ${}^q_{\kappa}I$ with

$$P_{P^q} = \{[L({}^q_{\kappa}I) = p_1^q, p_2^q - 1], [p_2^q, p_3^q - 1], \dots, [p_{m'}^q, R({}^q_{\kappa}I)]\}.$$

We depict each P_{P^q} in Fig. 4 at the very top in purple. Similar to Q , we can compute each threshold $p \in P^q$ iteratively by considering only the border values of all reachable transition guards from κ in Inequality 5. The worst case complexity to compute P_{P^q} is $O(|{}^q_{\kappa}I| \cdot |L| \cdot |\Delta|)$. For all $z \in {}^q_{\kappa}I^p \in P_{P^q}$, we have

$$\varepsilon\text{-CLO}(\delta_{\mathcal{A}}(\xi(\langle \kappa, z \rangle), \sigma)) = \varepsilon\text{-CLO}(\delta_{\mathcal{A}}(\xi(\langle \kappa, L({}^q_{\kappa}I^p) \rangle), \sigma)) \oplus (z - L({}^q_{\kappa}I^p)).$$

We can therefore represent the transitions of $f_{\Delta}(e, \sigma)$ in ${}^q_{\kappa}I^p$ by the transition

$$(\kappa, \sigma, {}^q_{\kappa}I^p, \perp, t, \kappa'), \text{ with } \xi(\langle \kappa', t \rangle) = \delta_{\mathcal{A}}(\xi(\langle \kappa, L({}^q_{\kappa}I^p) \rangle), \sigma). \quad (6)$$

5.2 Deriving Threshold Estimators from Timed Automata

Algorithm 1 combines the techniques described in the previous section in a concise procedure to efficiently derive TE from TA. We first compute the initial location of κ_0 (Line 1). We then initialize the set of newly discovered locations \mathcal{U} and add κ_0 (Line 2). For all newly discovered locations κ , we compute $\tau\text{-}\varepsilon\text{-CLO}(\kappa)$ and η_{κ} , and set the delay transition $\delta_{\tau}(\kappa, \eta_{\kappa}) := \langle \kappa', t' \rangle$ (Line 5). We then compute the observable transitions (Line 6–18). For this purpose, we first compute the partition P_Q to identify all intervals in κ that enable the same transitions. Secondly, we compute all observable transitions for each interval in P_Q (Line 8–17). We check if Case 1 or Case 2 from the construction in Sect. 5.1 apply (Line 9 and 11). If none of the cases apply, we use the construction, described in the proof of Theorem 1.

Algorithm 1 always terminates, since there exist only finitely many states to visit under a time abstraction α . The worst case complexity of a single iteration is $O(|\Sigma| \cdot M_{\mathcal{A}} \cdot |L| \cdot |\Delta|)$, since $\eta_{\kappa} \leq M_{\mathcal{A}}$ for any $\kappa \in K$. In the worst case, we get $|K| = 2^{|L| \cdot M_{\mathcal{A}}}$. The total worst case complexity of Algorithm 1 is therefore $O(2^{|L| \cdot M_{\mathcal{A}}} \cdot |\Sigma| \cdot M_{\mathcal{A}} \cdot |L| \cdot |\Delta|)$.

Example 7. Consider TA \mathcal{A}_E (Fig. 3a). Figure 3c depicts an equivalent TE \mathcal{E}_E that can be computed with Algorithm 1. Note that \mathcal{E}_E contains transitions with intervals that span over multiple time instances. Consider the transition $(\kappa_2, o, [0, 1], \top, 0, \kappa_0)$. This transition can be directly computed using the method described in Case 1. Similarly, the transition $(\kappa_0, o, [2, 3], \perp, 0, \kappa_2)$ can be directly computed using the method described in Case 2.

Algorithm 1. Synthesis of Threshold Estimators from Timed Automata.**Input:** TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$ and time abstraction α **Output:** TE $\mathcal{E} = (K, \kappa_0, \xi, \Sigma_o, \delta_\tau, \Delta_\mathcal{E})$

```

1: compute initial location  $\kappa_0 \leftarrow \varepsilon\text{-CLO}(\{\langle \ell, 0 \rangle \mid \ell \in L_0\})$ 
2: initialize set  $\mathcal{U}$  and add  $\kappa_0$ 
3: while  $\mathcal{U}$  is not empty do
4:    $\kappa \leftarrow$  extract an element from  $\mathcal{U}$ 
5:   compute  $\tau\text{-}\varepsilon\text{-CLO}(\kappa)$  and  $\eta_\kappa$  and set  $\delta_\tau(\kappa, \eta_\kappa) := \langle \kappa', t' \rangle$ 
6:   for each  $\sigma \in \Sigma_o$  do
7:      $P_Q \leftarrow$  compute partition from  $\tau\text{-}\varepsilon\text{-CLO}(\kappa)$  for  $\sigma$ 
8:     for each  ${}^q I \in P_Q$  do
9:       if Case 1 applies for  ${}^q I$  then
10:        compute all  $(\kappa, \sigma, {}^q I, \top, t, \kappa')$  (as in Eq. 4) and add to  $\Delta_\mathcal{E}$ 
11:       else if Case 2 applies for  ${}^q I$  then
12:        compute all  $(\kappa, \sigma, {}^q I^p, \perp, t, \kappa')$  (as in Eq. 6) and add to  $\Delta_\mathcal{E}$ 
13:       else
14:        compute all  $(\kappa, \sigma, [z, z], \top, 0, \kappa')$  (as in Eq. 2) and add to  $\Delta_\mathcal{E}$ 
15:       end if
16:       add all newly discovered  $\kappa'$  in  $\mathcal{U}$ 
17:     end for
18:   end for
19: end while

```

6 Verification of Current-State Opacity

In this section, we discuss how *threshold estimators* (TE) can be used to verify *timed automata* (TA). From a given TE, we could simply construct an equivalent τ -DFA that enumerates all inferred states of the TE. This would enable all FA-based verification methods for any TA. However, the scalability of such a method would again be limited by the complexity of τ -DFA [10]. It would therefore be more efficient to directly apply verification methods on TE to avoid the state explosion caused by τ -DFA. In the following we illustrate such direct application for the verification of *current-state opacity* (CSO) as an example.

Theorem 2. *Given a TA $\mathcal{A} = (L, L_0, \Sigma, \Sigma_o, \Delta)$, let $\mathcal{E} = (K, \kappa_0, \xi, \Sigma_o, \delta_\tau, \Delta_\mathcal{E})$ be a TE, derived from \mathcal{A} . Further, let $L_S \subseteq L$ be a set of secret locations, let $Q := L \times \mathbb{N}$ be the set of all states in \mathcal{A} , and let $Q_S := L_S \times \mathbb{N}$ be the set of all secret states in \mathcal{A} . CSO holds for \mathcal{A} if there exists no $\kappa \in K$, such that $\xi(\langle \kappa, t \rangle) \cap (Q \setminus Q_S) = \emptyset$ for $t \in [0, \eta_\kappa - 1]$.*

Proof. By Definition 5, \mathcal{A} is CSO if $\delta_\mathcal{A}(\{\langle \ell_0, 0 \rangle \mid \ell_0 \in L_0\}, w) \cap (Q \setminus Q_S) \neq \emptyset$ holds for any word $w \in (\tau^* \Sigma_o)^*$. Notice that for any $e \in \mathcal{P}(L \times \mathbb{N})$ we have $\{\ell \mid \langle \ell, t \rangle \in e\} = \{\ell \mid \langle \ell, t \rangle \in \alpha(e)\}$ for a time abstraction α . This implies that we only need to prove $\alpha(\delta_\mathcal{A}(\{\langle \ell_0, 0 \rangle \mid \ell_0 \in L_0\}, w)) \cap (Q \setminus Q_S) \neq \emptyset$. Further, by Theorem 1, we can equivalently prove $\xi(\delta_\mathcal{E}(\langle \kappa_0, 0 \rangle, w)) \cap (Q \setminus Q_S) \neq \emptyset$. $\{\langle \kappa, t \rangle \mid \kappa \in K, t \in [0, \eta_\kappa - 1]\}$ is the set of all reachable states in \mathcal{E} . If there exists no $\kappa \in K$, such that $\xi(\langle \kappa, t \rangle) \cap (Q \setminus Q_S) = \emptyset$, then $\xi(\delta_\mathcal{E}(\langle \kappa_0, 0 \rangle, w)) \cap (Q \setminus Q_S) \neq \emptyset$ must hold for any $w \in (\tau^* \Sigma_o)^*$.

Intuitively, CSO can be verified by checking whether a TE contains a CSE that only consists of secret states. This can be done during the construction of TE. The computation of TE is therefore an efficient method to verify CSO of TA. Since TE and τ -DFA are equally expressive, any FA-based verification method can be adapted to TE. Moreover, since TE are more compact, it is likely that such an adapted verification method is more efficient as in the case of CSO.

Example 8. Consider the example TA \mathcal{A}_E (Fig. 3a). Assume ℓ_2 is the only secret location with $L_S := \{\ell_2\}$. Now consider the TE \mathcal{E}_E , derived from \mathcal{A} (Fig. 3c). CSO does not hold, since for example κ_2 exists with $\xi(\langle \kappa_2, 0 \rangle) = \{\langle \ell_2, 2 \rangle\}$ and $\{\langle \ell_2, 2 \rangle\} \cap (Q \setminus Q_S) = \emptyset$. Intuitively, this means that there exists a word $w = \tau\tau o$, that leads to the CSE $\{\langle \ell_2, 2 \rangle\}$, which contains only a secret state. This means that an intruder would now know that the secret location ℓ_2 must be active. Equivalently, $\{\langle \ell_2, 2 \rangle\}$ is also the active state in the τ -DFA A_E (Fig. 3b) after reading w .

7 Evaluation

In this paper, we present a new method to efficiently compute estimators for *discrete-timed automata* (TA). The key motivation is to compute estimators efficiently that are expressive enough to represent the deterministic behavior of any TA, like *deterministic tick automata* (τ -DFA) [9]. For this purpose, we introduce *threshold estimators* (TE), a compact estimator model that can be efficiently derived from TA with Algorithm 1. To evaluate our method, we compare it to the computation of estimators using τ -DFA as proposed in [10]. We have implemented a software prototype¹ that derives TE and τ -DFA from a given TA. For comparison, we use the size and the computation time of the resulting estimators. As input systems, we use 11 diverse case studies, taken from the literature [1, 2, 5, 10, 11, 20, 22–24]. All tests were performed on an Intel Core i5-10600K CPU with 48 GB of RAM.

In the following, we first introduce the case studies in Sect. 7.1. Afterwards, we define the used metrics in our experiments in Sect. 7.2. Finally, we present the results of our experiments in Sect. 7.3.

7.1 Case Studies

We use 11 diverse case studies from different domains to test the applicability of our method. The key characteristics of our case studies are summarized in Table 1, listing the number of locations ($|L|$), the number of transitions ($|\Delta|$), the alphabet size ($|\Sigma|$), and the largest constant in a guard (M).

AKM, TCP, and SCTP are network protocols. SCTP is a model of a receiver for the *stream control transmission protocol* [22]. AKM is a model of an Android Wi-Fi authentication system [24], and TCP is a model of the connection management of the *transmission control protocol* [20].

¹ Available at <https://gitlab.com/julianklein/threshold-estimation>.

Table 1. Overview of the case studies.

	AKM	TCP	SCTP	PC	CAS	SCHED	OVEN	HVAC	WSN	WSNET	MED
$ L $	4	11	41	8	8	23	89	11	63	25	8
$ \Delta $	18	19	155	24	17	28	179	41	185	50	9
$ \Sigma $	12	11	19	16	12	8	9	22	6	9	8
M	2500	240	1000	10	27000	15	5000	2000	300000	30	10

PC, CAS, SCHED, OVEN, and HVAC represent time-sensitive embedded systems. In particular, PC and CAS come from the automotive domain. PC is a *particle counter*, measuring the number of particles in exhaust gases [2] and CAS is a *car alarm system* [1]. SCHED represents a *schedule* of tests for integrated circuits [5]. OVEN and HVAC are smart home devices. OVEN is a model of an *oven* with time-controlled baking program [11]. HVAC models a *heating ventilation air conditioning* unit, adapted from [23].

WSN, WSNET, and MED resemble applications of network architectures. WSN is a model of a single *wireless sensor node* that collects data at a fixed timed rate [11]. WSNET models a *wireless sensor network* that locates an agent in a large area [10]. MED is a model of a *medical cloud service* that processes patient data [10].

7.2 Metrics

In order to assess the efficiency of our presented method, we measure the computation times and calculate the sizes of the obtained estimators. We average the computation times over 100 runs to account for runtime variations due to operating scheduling and hardware effects. For each computation, we set a timeout of 60 seconds. To assess the memory usage of the estimators, we define a function *size* for both τ -DFA and TE. Intuitively, *size* correlates with the memory, required to represent an estimator in a real machine. *size* is defined as follows.

Size of a τ -DFA (Definition 4): Generally, in a CSE $e \in \mathcal{P}(L \times \mathbb{N})$, we store a location ($\ell \in L$) and a time value ($t \in \mathbb{N}$) for each $\langle \ell, t \rangle \in e$. We therefore define $size(e) = |e| \cdot 2$. Each state $e \in E$ in a τ -DFA is a CSE and has therefore a size of $size(e)$. Each transition $e \xrightarrow{\sigma} e'$ stores two state references ($e, e' \in E$) and one symbol ($\sigma \in \Sigma$). We therefore have $size(\delta_A) = |E| \cdot |\Sigma| \cdot 3$. In total, the size of a τ -DFA A is given by

$$size(A) = \sum_{e \in E} size(e) + |\Sigma| + size(\delta_A).$$

Size of TE (Definition 7): Each location $\kappa \in K$ stores a CSE e and therefore has a size of $size(e)$. Any $\xi(\langle \kappa, t \rangle)$ can be directly computed from τ - ε -CLO(κ). We therefore set $size(\xi) := \sum_{\kappa \in K} size(\tau$ - ε -CLO(κ)). For each $(s_i, e_i) \in \tau$ - ε -CLO(e), we store a time value ($s \in \mathbb{N}$) and a CSE (e_i). We thus get $size(\tau$ - ε -CLO(e)) = $\sum_{(s_i, e_i) \in \tau$ - ε -CLO(e)} $size(e_i) + 1$. For each $\kappa \in K$, we store one delay transition

Table 2. Calculated estimator sizes in the experiments.

System	α_L			α_R		
	TE	τ -DFA	%(TE, τ -DFA)	TE	τ -DFA	%(TE, τ -DFA)
AKM	15249	102283	-85.09%	45249	44087783	-99.9%
TCP	1045	66333	-98.42%	6085	584973	-98.96%
SCTP	164829	1948205	-91.54%	171829	-	-
PC	354	2535	-86.04%	354	6835	-94.82%
CAS	171289	9266477	-98.15%	-	-	-
SCHED	572	5183	-88.96%	572	9008	-93.65%
OVEN	2352	215265	-98.91%	2352	4002765	-99.94%
HVAC	64631	24985303	-99.74%	70631	-	-
WSN	902112	-	-	-	-	-
WSNET	2961	27169	-89.1%	4323	71674	-93.97%
MED	211	1401	-84.94%	211	2861	-92.62%

$\langle \kappa, t \rangle \xrightarrow{\tau} \langle \kappa', t' \rangle$, consisting of two location references ($\kappa, \kappa' \in K$), and two time values ($t, t' \in \mathbb{N}$). We therefore get $size(\delta_\tau) = |K| \cdot 4$. Each $(\kappa, \sigma, [a, b], \lambda, z, \kappa') \in \Delta_{\mathcal{E}}$ stores two location references ($\kappa, \kappa' \in L$), one symbol ($\sigma \in \Sigma$), three time values ($a, b, z \in \mathbb{N}$), and one time modifier ($\lambda \in \{\top, \perp\}$). We therefore get $size(\Delta_{\mathcal{E}}) = |\Delta_{\mathcal{E}}| \cdot 7$. In total, the size of a TE \mathcal{E} is given by

$$size(\mathcal{E}) = \sum_{\kappa \in K} size(\kappa) + size(\xi) + |\Sigma| + size(\delta_\tau) + size(\Delta_{\mathcal{E}}).$$

7.3 Experiments

We now present the results of our experiments. In each experiment, we compute the TE and τ -DFA from a given TA. Both methods are depicted in Fig. 1. We compute TE using Algorithm 1. To compute τ -DFA, we first derive an equivalent τ -FA [10], which we then determinize using the standard subset construction [19]. Both methods require a time abstraction α . We test both the standard region abstraction α_R [3], and the local time abstraction α_L [10], to assess the impact of the selected time abstraction. The calculated sizes of the estimators are displayed in Table 2 and the measured computation times are presented in Table 3. Tables 2 and 3 depict the results for α_L (left) and α_R (right) for the derived TE (our new method), τ -DFA (the method in [10]), and the percentage improvement $\%(TE, \tau\text{-DFA}) := 100 \cdot (\frac{TE}{\tau\text{-DFA}} - 1)$.

Considering the estimator sizes in Table 2, we see a significant improvement in all systems. The computations using α_L testify improvements ranging between -85% and -99% . The improvements are even more significant for α_R , ranging between -90% and -99% . Regarding only the sizes of the τ -DFA, we also see a significant decrease with α_L , compared to α_R . Our results thereby match the

Table 3. Measured computation times in the experiments.

System	α_L			α_R		
	TE	τ -DFA	%(TE, τ -DFA)	TE	τ -DFA	%(TE, τ -DFA)
AKM	42.57 ms	43.83 ms	-2.86%	18.51 s	32.53 s	-43.11%
TCP	16.25 ms	16.36 ms	-0.65%	0.18 s	0.29 s	-38.13%
SCTP	0.66 s	1.76 s	-62.11%	21.62 s	-	-
PC	0.22 ms	3.27 ms	-92.99%	0.21 ms	3.43 ms	-93.73%
CAS	4.50 s	6.43 s	-29.96%	-	-	-
SCHED	0.34 ms	4.29 ms	-91.98%	0.30 ms	4.12 ms	-92.71%
OVEN	1.77 ms	1.51 s	-99.88%	1.09 ms	11.51 s	-99.99%
HVAC	6.25 s	36.92 s	-83.06%	16.70 s	-	-
WSN	11.94 s	-	-	-	-	-
WSNET	7.81 ms	10.57 ms	-26.08%	13.05 ms	28.16 ms	-53.65%
MED	0.25 ms	0.32 ms	-22.55%	0.85 ms	3.68 ms	-76.81%

results in [10]. Regarding the sizes of the TE with respect to the time abstractions, the results vary. For PC, SCHED, and OVEN the TE had the same sizes for α_L and α_R . For SCTP and HVAC, we see insignificant improvements for α_L . For AKM, TCP, and WSNET, we see significant improvements, when α_L is used. For CAS and WSN, the computation of the TE timed out for α_R , indicating a significant improvement for α_L .

Considering the computation times in Table 3, we see a significant improvement for most systems with α_L . The computation times of the TE for AKM and TCP are only slightly reduced by -2.86% and -0.65% respectively. For CAS, WSNET, and MED, we see reductions between -20% and -30% . All other systems showed improvements of at least -60% , going up to -99% for OVEN. The computation of the τ -DFA of WSN timed out, meaning it took at least 60 s, while we were able to compute a TE in less than 12 s. Considering α_R , the improvements of our TE model are even more significant. For the systems that did not time out, we see improvements ranging between -38% and -99% . When comparing the impact of the abstractions on the TE computations, we see that α_L performs better for most systems. For PC, SCHED, and OVEN, α_L performs slightly worse, as the size of the TE does not change for both time abstractions (see Table 2), and the computation of α_L is more expensive than the computation of α_R [10].

Overall, our results validate that the use of our TE model provides significant improvements in both size and computation time, regardless of the time abstraction. Moreover, they show that the use of α_L further improves the efficiency of our presented method significantly for most evaluated systems. Using Algorithm 1 with α_L , we were able to compute TE for all 11 systems in less than one minute. We thereby demonstrate that our TE model provides an efficient state estimation method for TA in practice.

8 Conclusion

In this paper, we have addressed the state estimation problem of *discrete-timed automata* (TA). We have introduced *threshold estimators* (TE), a compact estimator model for TA. Moreover, we have provided a method to efficiently derive TE from TA. We have further also discussed how TE can be used to verify TA and demonstrated the direct verification of *current-state opacity* (CSO) of TA as an example. To evaluate our method we have implemented a software prototype and compared its performance to the computation of estimators with *deterministic tick automata* (τ -DFA) [10]. In our experiments, we have tested both methods on 11 realistic case studies from multiple domains. Our evaluation shows that our new method significantly reduces the computational costs to derive estimators from all evaluated systems.

In the future, we plan to adapt verification methods to the TE model to also verify other state-based opacity notions and state inference properties, that have been shown to be verifiable using estimators [28, 29]. Moreover, we plan to also explore possibilities to adapt estimator-based opacity enforcement methods to TE [26].

References

1. Aichernig, B.K., Lorber, F., Ničković, D.: Time for mutants — model-based mutation testing with timed automata. In: Veanes, M., Viganò, L. (eds.) Tests and Proofs, pp. 20–38. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38916-0_2
2. Aichernig, B.K., Pferscher, A., Tappler, M.: From passive to active: learning timed automata efficiently. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11–15, 2020, Proceedings, pp. 1–19. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_1
3. Alur, R., Dill, D.: The theory of timed automata. In: de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G. (eds.) Real-Time: Theory in Practice, pp. 45–73. Springer, Berlin, Heidelberg (1992). <https://doi.org/10.1007/BFb0031987>
4. Alur, R., Madhusudan, P.: Decision problems for timed automata: a survey. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems, pp. 1–24. Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_1
5. An, J., Zhan, B., Zhan, N., Zhang, M.: Learning nondeterministic real-time automata. ACM Trans. Embed. Comput. Syst. (2021). <https://doi.org/10.1145/3477030>
6. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T.: When are timed automata determinizable? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikolettseas, S., Thomas, W. (eds.) Automata, Languages and Programming, pp. 43–54. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02930-1_4
7. Bertrand, N., Stainer, A., Jéron, T., Krichen, M.: A game approach to determinize timed automata. Formal Methods Syst. Design (2015). <https://doi.org/10.1007/s10703-014-0220-1>

8. Choffrut, C., Goldwurm, M.: Timed automata with periodic clock constraints. *J. Autom. Lang. Combinatorics* (2000). <https://doi.org/10.25596/jalc-2000-371>
9. Gruber, H., Holzer, M., Kiehn, A., König, B.: On timed automata with discrete time – structural and language theoretical characterization. In: De Felice, C., Restivo, A. (eds.) *Developments in Language Theory*, pp. 272–283. Springer, Berlin, Heidelberg (2005). https://doi.org/10.1007/11505877_24
10. Klein, J., Kogel, P., Glesner, S.: Verifying opacity of discrete-timed automata. In: *International Conference on Formal Methods in Software Engineering*. Association for Computing Machinery (2024). <https://doi.org/10.1145/3644033.3644376>
11. Kogel, P., Klös, V., Glesner, S.: Correction to: learning mealy machines with local timers. In: Li, Y., Tahar, S. (eds.) *Formal Methods and Software Engineering: 24th International Conference on Formal Engineering Methods, ICFEM 2023, Brisbane, QLD, Australia, November 21–24, 2023, Proceedings*, pp. C1–C1. Springer Nature Singapore, Singapore (2023). https://doi.org/10.1007/978-981-99-7584-6_23
12. Komenda, J., Lefebvre, D.: On tick automata for distributed timed DESs with synchronisations and minimal time constraints. *IFAC-PapersOnLine* (2023). <https://doi.org/10.1016/j.ifacol.2023.10.039>
13. Lai, A., Lahaye, S., Giua, A.: State estimation of max-plus automata with unobservable events. *Automatica* (2019). <https://doi.org/10.1016/j.automatica.2019.03.003>
14. Lai, A., Lahaye, S., Giua, A.: Verification of detectability for unambiguous weighted automata. *IEEE Trans. Autom. Control* (2020). <https://doi.org/10.1109/TAC.2020.2995173>
15. Lawford, M.S.: *Model Reduction of Discrete Real-Time Systems*. University of Toronto (1998)
16. Li, J., Lefebvre, D., Hadjicostis, C.N., Li, Z.: Observers for a class of timed automata based on elapsed time graphs. *IEEE Trans. Autom. Control* (2021). <https://doi.org/10.1109/TAC.2021.3064542>
17. Lin, L., Su, R., Brandin, B.A., Ware, S., Zhu, Y., Sun, Y.: Synchronous composition of finite interval automata. In: *2019 IEEE 15th International Conference on Control and Automation*. IEEE (2019). <https://doi.org/10.1109/ICCA.2019.8899529>
18. Lorber, F., Rosenmann, A., Ničković, D., Aichernig, B.K.: Bounded determinization of timed automata with silent transitions. *Real-Time Syst.* (2017). <https://doi.org/10.1007/s11241-017-9271-x>
19. Noord, G.V.: Treatment of epsilon moves in subset construction. *Comput. Linguist.* (2000). <https://doi.org/10.1162/089120100561638>
20. Postel, J.: *Transmission Control Protocol*. RFC 793 (1981). <https://doi.org/10.17487/RFC0793>, <https://www.rfc-editor.org/info/rfc793>
21. Saboori, A., Hadjicostis, C.N.: Opacity-enforcing supervisory strategies via state estimator constructions. *IEEE Trans. Autom. Control* (2011). <https://doi.org/10.1109/TAC.2011.2170453>
22. Stewart, R.: *Stream control transmission protocol*. Tech. rep. (2007)
23. Taylor, J.T., Taylor, W.T.: *Patterns in the Machine: A Software Engineering Guide to Embedded Development*. Apress, Berkeley, CA (2021). <https://doi.org/10.1007/978-1-4842-6440-9>
24. Vaandrager, F., Ebrahimi, M., Bloem, R.: Learning mealy machines with one timer. *Inf. Comput.* (2023). <https://doi.org/10.1016/j.ic.2023.105013>
25. Wu, Y.C., Lafortune, S.: Comparative analysis of related notions of opacity in centralized and coordinated architectures. *Discrete Event Dyn. Syst.* (2013). <https://doi.org/10.1007/s10626-012-0145-z>

26. Wu, Y.C., Lafortune, S.: Synthesis of insertion functions for enforcement of opacity security properties. *Automatica* (2014). <https://doi.org/10.1016/j.automatica.2014.02.038>
27. Zhang, K.: Detectability of labeled weighted automata over monoids. *Discrete Event Dyn. Syst.* (2022). <https://doi.org/10.1007/s10626-022-00362-8>
28. Zhang, K.: A unified concurrent-composition method to state/event inference and concealment in labeled finite-state automata as discrete-event systems. *Ann. Rev. Control.* (2023). <https://doi.org/10.1016/j.arcontrol.2023.100902>
29. Zhang, K.: State-based opacity of labeled real-time automata. *Theoret. Comput. Sci.* (2024). <https://doi.org/10.1016/j.tcs.2023.114373>



LRNN: A Formal Logic Rules-Based Neural Network for Software Defect Prediction

Yuxiang Shang and Shaoying Liu^(✉)

Hiroshima University, Hiroshima, Japan
shangyuxiang@hiroshima-u.ac.jp

Abstract. Class imbalance in deep learning systems can lead to biased models due to the unbalanced distribution of data classes in the training dataset. When such biased models are adopted for software defect prediction (SDP), the prediction precision will be significantly affected. To improve the mechanism of existing related deep learning models for SDP, we propose a new neural network, called Logic Rules Neural Network (LRNN), to address the class imbalance problem. LRNN is characterized by utilizing formal logic rules on the association between code metrics to improve the efficiency of deep learning. We describe how relevant formal logic can be identified and selected and how it can be utilized in the learning process. We evaluate the performance of our approach by conducting some experiments using the NASA, PROMISE, AEEEM repositories. The results demonstrate a significant improvement over several existing algorithms.

Keywords: Formal Logic · Neural Network · Defect Prediction

1 Introduction

In software engineering, predicting software defects (SDP) is fundamentally important [22]. As software becomes increasingly complex and large-scale, its quality has emerged as a widespread concern among both industrial and academic circles. Manual debugging, in terms of time and financial resources, proves to be prohibitively expensive to achieve reliability and validity of the software [19]. Therefore, the significance of automated defect mining is underscored.

Most existing deep neural networks for Software Defect Prediction (SDP) are designed specifically for detecting defects within a single project. In this context, the prediction model is trained using a segment of the project. Instances within the project can be assigned their class labels. Subsequently, this model is employed to forecast defects across the entire project. However, addressing a software project in this manner frequently encounters issues related to class imbalance [24]. This is easily understandable, as the volume of code without defects significantly outweighs the volume with defects. We analyzed multiple code repositories, including NASA [28], PROMISE [2], AEEEM [7], and identified a consistent issue of class imbalance across them.

Beyond the issue of class imbalance, we observed that an excessive number of features in the dataset also impacts the accuracy of predictions. Data preprocessing is very important, especially in high dimensional dataset [28]. The approach to data processing directly influences the accuracy of predictions. Principal Component Analysis (PCA) [31] serves as the primary method for reducing data dimensionality. However, this approach compromises the interpretability of the predictive model, as PCA alters the original features.

This paper aims to address the class imbalance issue and non-interpretability of data dimensionality reduction by integrating formal logic into neural networks. These logic rules are imparted to the neural network prior to training the machine learning system with historical data. In this case, the model requires only minimal training and can identify features associated with defective instances.

Motivated by these insights, we propose Logic Rules Neural Network (LRNN), a Neuro-symbolic AI system for feature selection with logic rule reasoning. This framework focuses on identifying a critical few targets to address class imbalance without sampling. It is a deep neural network classifier with two special dense layers. Initially, a rule learning model [11] establishes the initial layer of the LRNN network as a generator of formal logic rules, aimed at discovering data regularities that can be formulated as IF-THEN rules. Subsequently, Logical Neural Networks (LNN) [26] form the upper layer of the LRNN network, enabling both neural network-style learning and traditional AI-style reasoning concurrently. After reasoning with formal logic, code metrics related to defective samples are selected as attribution of the deep neural network classifier. In our study, we conducted extensive experiments on the NASA, PROMISE, and AEEEM datasets, employing various performance metrics to assess the effectiveness of our innovative model, which is elaborated in Sect. 4. This paper presents the following contributions:

- Introducing a new neural network model called LRNN, based on formal logic, this model can be trained effectively using a small amount of historical data for prediction tasks.
- To preserve information for the limited data instances, LRNN can mitigate the effects of class imbalance without over-sampling or under-sampling techniques.
- The LRNN model is evaluated through some experiments on multiple datasets, and it demonstrates interpretability in feature selection.

This article is organized as follows: Sect. 2 provides a concise review of related techniques in defect prediction. Section 3 delves into the algorithmic details of our LRNN method. Section 4 discusses the experimental results, and Sect. 5 concludes the paper.

2 Related Work

As neural networks gain popularity, their limitations are becoming increasingly apparent. The most critical among these issues is the reduced robustness and

interpretability of deep learning. Hence, the integration of domain knowledge into neural network modeling has gained widespread attention as a research challenge [5]. Neuro-symbolic AI [17] serves as a bridge between neural networks and formal logic rules.

First-order logic, representing the simplest method for knowledge representation, has been attempted for integration into machine learning. As a result, Inductive Logic Programming (ILP) [23] was proposed. The ILP [4] approach is not a black-box model that is difficult to analyze; it can make use of domain knowledge described in first-order logic. This approach significantly enhances the interpretability of machine learning models.

Recently, there's been an increasing fascination with neural networks that incorporate formal logic rules. Hu et al. [18] developed an iterative distillation method that infuses the structured information from formal logic into the weights of neural networks. Li [21] and Qu [25] attempted to integrate formal logic into various neural networks, including convolutional neural networks, recurrent neural networks, and deep neural networks, specifically targeting their input and output layers. IBM Research [9] proposed a neural network with reasoning capabilities, named Logical Neural Networks(LNN). LRNN presented in this paper is founded on this framework. The LNN converts formal logic rules into constraints that limit the activation function of the neural network, thus controlling the output of the model.

3 The LRNN Approach

Building on existing research into deep learning for defect prediction, we suggest an inferable neural network model informed by software code metrics. The architecture of the LRNN model is illustrated in Fig. 1. Initially, we partition the source software project into two segments randomly. One part is fed into a formal logic rules generator. The code in these software projects is analyzed and evaluated through code metrics. The “rules” refer to the range of values for code metrics and their interrelationships. Thus, the input for this module is the code’s metrics. We consider the formal logic rules generated by the rule learning module. Subsequently, a separate portion of the code is designated for the training and testing phases of the neural network.

The Logical Neural Network (LNN) [26] is engineered to integrate neural network-style learning and classical AI-style reasoning smoothly. This occurs because LNNs incorporate rules-based parameters to adjust the activation function’s threshold, thereby directly influencing the neural network’s output. LNN is used directly as a module in our proposed method to help reason about logical statements. Before the LNN layer, there is a rule generation layer that applies a rule learning technique, as detailed in Sect. 3.1. The input to this layer consists of all code metrics. After the LNN layer, an attribute selection layer follows. This layer uses the reasoning results from the LNN to filter code metrics related to defective samples, which serve as input attributes for the neural network classifier. By following this process, the rules are integrated with the neural network.

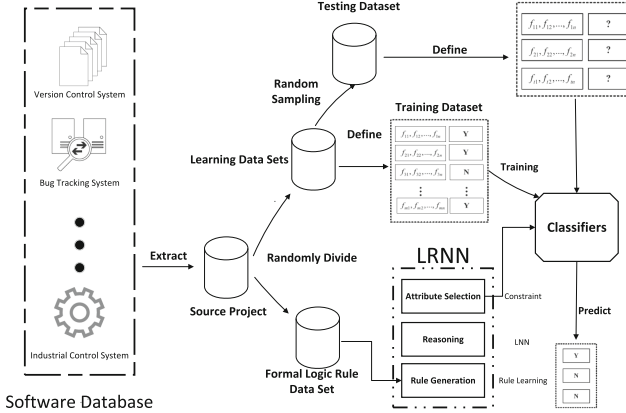


Fig. 1. Structure of LRNN Approach

In this context, it’s worth noting that these formal logic rules constitute a logical proposition regarding the association between code metrics and the target label (whether it is a defective sample or not). The LNN should infer specific code metrics based these logical propositions, deemed highly relevant to the defective sample, for use in predictions. These metrics are used exclusively as attributes, aiming to enhance the predictive power for defective instances. This is the attribute selection in the Fig. 1. Because it employs logic-based attribute selection, this approach is interpretable. Following this, the classifier will employ these defect-related code metrics as attributes within its binary classification process.

3.1 Rule Learning

In machine learning, a “rule” is a semantically clear pattern that describes the distribution of data. These rules are usually formulated as logical statements in the “IF...THEN...” format [11]. Rule learning entails deriving a set of rules from the training dataset to distinguish between various examples [33]. Below is an example of a rule.

$$\oplus \leftarrow f_1 \wedge f_2 \wedge \cdots \wedge f_L, \quad (1)$$

where \leftarrow is implication. \oplus is consequent, representing the result of this rule. The rule’s body comprises a conjunction of logical literals, denoted as f_k , with the conjunction symbol “ \wedge ” representing “and”. Rule learning covers two main areas of study [12]. Descriptive rule learning aims at identifying regularities within specific parts of the dataset, while predictive rule learning seeks to generalize from the dataset for making predictions on new data. This paper concentrates on descriptive rule learning.

```

function FINDPREDICTIVERULE (Examples)

Input: Examples, a set of positive and negative examples for a class c.

//initialize the rule body
 $rb \leftarrow \emptyset$ 
// repeatedly find the best refinement
repeat
    build refinements  $R \leftarrow \{rb' \mid rb' = rb \wedge f, \text{ for some feature } f\}$ 
    evaluate all  $rb' \in R$  according to some quality criterion
     $rb = \text{the best refinement in } R$ 
until  $rb$  satisfies a stopping criterion
    or covers no examples

Output: rule ( $c \leftarrow R$ )

```

Fig. 2. Greedy search for predictive rules

Generation of Formal Logic Rules. In descriptive rule discovery, the primary focus is on discovering regularities that are observable within a given dataset. The most commonly used algorithm is the Top-Down Hill-Climbing Algorithm [10]. In Fig. 2, a greedy hill-climbing algorithm is illustrated for discovering a single predictive rule. Beginning with an empty rule body, it progressively introduces new conditions. When incorporating a condition, the algorithm explores all potential additions and assesses them using a heuristic quality criterion. This criterion typically depends on the number of examples that are covered and uncovered, distinguishing whether they belong to class “c” (positive examples) or do not belong to class “c” (negative examples). Here, conditions are added in an iterative manner to the rule until it no longer includes any positive examples. Eventually, a set of rules in this given dataset is established.

The technique for generating a specific piece of rule is the Ripper algorithm [3] in this paper. To develop a rule, we utilize a training set comprising both positive and negative examples. Consistent with class “c” in the greedy hill-climbing algorithm.

Table 1. Data Format

Metric_1	Metric_2	Metric_n	Class
Attrib1	Attrib2	Attribn	Defective
Attrib1	Attrib2	Attribn	Defect-free

Here is a simple example to illustrate how a single rule is generated. The dataset’s structure, from which the rules were derived, is depicted in Table 1. Code metrics serve as attributes for the data. Given a sufficient dataset in the specified format, RIPPER can extract rules as follows:

- Defective (class1):- $\text{Attrib1} = x, \text{Attrib5} = y.$
- Defect-free (class2):- $\text{Attrib2} = z.$
- None (class3):- true.

where if **Attrib1** is x, and **Attrib5** is y, then this sample belongs to “class1”. If **Attrib2** is z, then this sample belongs to “class2”; “class3” is the default class; that is, a sample is classified as “class3” if it fails to meet any of the previously mentioned rules. This process generates a rule for each data point, which is then added to the rule set.

Foil gain is utilized to quantify the extent of data extraction into rules. The expression for foil gain is as follows:

$$\text{FoilGain} = \hat{n}_+ \times \left(\log_2 \frac{\hat{n}_+}{\hat{n}_+ + \hat{n}_-} - \log_2 \frac{n_+}{n_+ + n_-} \right) \quad (2)$$

where \hat{n}_+, \hat{n}_- represents the counts of positive and negative examples, respectively, covered by the new rules after incorporating candidate data; n_+, n_- denotes the counts of positive and negative examples covered by the original rules.

Based on greedy hill-climbing and ripper algorithms, for any given dataset, the number of generated rules is expected to closely approximate the dataset size. In other words, a rule is likely to be generated for each data. Without intervention, the method will continue generating rules indefinitely. This kind of overfitting is unacceptable. To prevent overfitting, adopting rule refinement is necessary.

Refinement of Formal Logic Rules. Pruning is crucial for mitigating overfitting resulting from the use of greedy algorithms. Likelihood Ratio Statistics (LRS) are employed to evaluate the quality of the generated rules. The formula for this statistic is presented below:

$$\text{LRS} = 2 \times \left(\hat{m}_+ \log_2 \left(\frac{\hat{m}_+}{\hat{m}_+ + \hat{m}_-} \right) + \hat{m}_- \log_2 \left(\frac{\hat{m}_-}{\hat{m}_+ + \hat{m}_-} \right) \right), \quad (3)$$

where m_+, m_- denote the number of positive and negative examples in the training set, respectively. \hat{m}_+, \hat{m}_- denote the number of positive and negative examples in the formal logic rules set, respectively. The LRS quantifies the discrepancy between the data distribution in the training set and the rules set. A larger LRS suggests that more rules can be derived from the dataset; conversely, a smaller LRS indicates a lack of regularity in this dataset. In addition to this, we have refined the generated rules using Reduced Error Pruning [13]. The number of generated rules can be regulated using the Likelihood Ratio Statistics (LRS).

In traditional rule learning, the refinement metric mentioned above is considered hyperparameter, meaning it is manually set. The accuracy of the prediction is directly influenced by the setting of this parameter, which is the primary limitation of using rule learning for prediction. In our approach, rule learning functions as a module for generating logical rules, while the neural network is

responsible for making predictions. This design effectively mitigates the limitations of rule learning and enhances the model’s robustness. In Experiment section, the rule learning model directly uses the Python version of RuleKit [15]. The RuleKit package is a rule-learning toolkit designed for prediction, classification, and survival analysis, developed by Gudy et al. [14].

3.2 Logical Neural Networks

Recently, due to the increasing complexity of deep learning, there has been a trend towards the development of interpretable models [16]. Although linear classifiers and decision trees are often viewed as interpretable, employing rules in first-order logic (FOL) results in a significantly more potent framework. In the process of learning these rules, neuro-symbolic AI commonly replaces conjunctions (and operations) and disjunctions (or operations) with differentiable t-norms and t-conorms, respectively [8]. However, due to these norms lacking learnable parameters, their behavior remains fixed, which constrains their capacity of accurately modeling the data.

Logical neural networks (LNN) [26] provide operators with parameters, enabling a more effective learning process from the data. To preserve the precise semantics of first-order logic (FOL), LNNs implement constraints during the learning of operators like conjunction. LNN- \wedge is expressed as:

$$\max(0, \min(1, \beta - \omega_1(1 - x) - \omega_2(1 - y))) \quad (4)$$

$$\text{subject to : } \beta - (1 - \alpha)(\omega_1 + \omega_2) \geq \alpha \quad (4a)$$

$$\beta - \alpha\omega_1 \leq 1 - \alpha \quad (4b)$$

$$\beta - \alpha\omega_2 \leq 1 - \alpha \quad (4c)$$

$$\omega_1, \omega_2 \geq 0$$

where $x, y \in [0, 1]$ are inputs, $\alpha \in [\frac{1}{2}, 1]$ is a hyperparameter and $\beta, \omega_1, \omega_2$ are learnable parameters. Note that $\max(0, \min(1, \cdot))$ clamps the output of LNN- \wedge between 0 and 1 regardless of $\beta, \omega_1, \omega_2, x$ and y . The constraints are most noteworthy. While Boolean conjunction only returns 1 or *TRUE* when both inputs are 1, LNNs relax this condition by using α as a proxy for 1 (conversely, $1 - \alpha$ as a proxy for 0). Specifically, Equation (4a) ensures that the output from the LNN exceeds the value α provided that its input also exceeds α . Similarly, Eqs. (4b) and (4c) limit the inputs to the LNN to one high and one low. For instance, Constraint (4b) forces the output of LNN- \wedge to be less than $1 - \alpha$ for $y = 1$ and $x \leq 1 - \alpha$. This formulation allows for unconstrained learning when $x, y \in [1 - \alpha, \alpha]$. By adjusting the value of α , users can manipulate the extent of learning; increasing α expands the area of unrestricted learning, while decreasing it narrows this region. Figure 3 depicts product t-norm and LNN- \wedge ($\alpha = 0.7$).

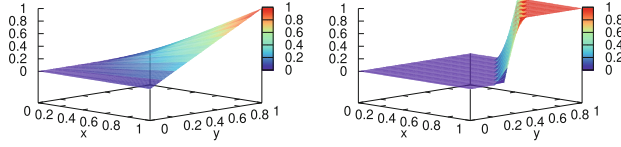


Fig. 3. (left) Product t -norm. (right) LNN- \wedge ($\alpha = 0.7$)

Traditional backpropagation faces difficulties with linear inequality constraints, such as Constraint (4a). However, the LNN framework includes specialized learning algorithms specifically tailored to tackle this challenge. The details about the framework can be found in the work by Riegel et al. [26].

Table 2. Samples of Formal Logic Rules

Rules for code metrics	Content
Rule1	IF LOC COMMENTS = $\langle 9.86, \text{inf} \rangle$ THEN label = {1}
Rule2	IF HALSTEAD ERROR EST = $\langle 0.55, \text{inf} \rangle$ THEN label = {1}
Rule3	IF CYCLOMATIC COMPLEXITY = $(-\text{inf}, 3.29)$ THEN label = {0}
Rule4	IF CYCLOMATIC COMPLEXITY = $\langle 11.4, 24.2 \rangle$ AND NUM UNIQUE OPERATORS = $\langle 14.57, \text{inf} \rangle$ AND HALSTEAD ERROR EST = $\langle 0.50, \text{inf} \rangle$ THEN label = {1}
Rule5	IF CYCLOMATIC COMPLEXITY = $(-\text{inf}, 8.64)$ AND NUM UNIQUE OPERATORS = $\langle 7.42, 10.0 \rangle$ AND HALSTEAD ERROR EST = $(-\text{inf}, 0.45)$ THEN label = {0}
Rule6	IF CYCLOMATIC COMPLEXITY = $(12.0, \text{inf})$ AND NUM UNIQUE OPERATORS = $(19.00, \text{inf})$ THEN label = {1}
Rule7	IF ESSENTIAL COMPLEXITY = $\langle 8.50, \text{inf} \rangle$ AND CYCLOMATIC COMPLEXITY = $\langle 32.50, \text{inf} \rangle$ AND LOC TOTAL = $\langle 78.60, \text{inf} \rangle$ AND LOC EXECUTABLE = $\langle 54.73, 68.50 \rangle$ AND HALSTEAD LENGTH = $(-\text{inf}, 249.78)$ AND HALSTEAD ERROR EST = $(-\text{inf}, 0.50)$ AND HALSTEAD EFFORT = $(1.22\text{e}+05, \text{inf})$ THEN label = {1}
Rule8	IF ESSENTIAL COMPLEXITY = $\langle 1.30, \text{inf} \rangle$ AND CYCLOMATIC COMPLEXITY = $(-\text{inf}, 2.45)$ AND LOC COMMENTS = $\langle 4.95, 5.84 \rangle$ AND LOC EXECUTABLE = $(-\text{inf}, 9.65)$ AND HALSTEAD LENGTH = $(-\text{inf}, 61.08)$ AND HALSTEAD ERROR EST = $(-\text{inf}, 0.50)$ THEN label = {0}
Rule9	

3.3 Simple Example

To show how reasoning works, here's a simple example. This example was generated using the NASA [28] dataset, and similar examples can be created for other datasets. Initially, we generated several rules using the rule learning approach, as detailed in Table 2. Within these rules, a label of 1 indicates defective

software, whereas a label of 0 denotes a non-defective project. Moreover, defective samples are termed as positive samples. At the same time, LNN model is employed to construct the predicates required for reasoning. The architecture of the LNN model is depicted in Fig. 4. There are three types of relationships for code metrics: *Association*, *Useful Metrics* and *Positive Metrics*. The assumption suggests that useful metrics include both positive and negative types, and metrics associated with positive types are also likely to be positive. *Association* represents the presence of a connection between two metrics. For instance, the repeated appearance of a metric combination across various rules suggests a connection between these metrics. *Useful Metrics* represents metrics distinctly linked to labels, including both positive and negative labels. *Positive Metrics* denotes the metric directly linked to the positive label, which is the centerpiece of our analysis. Our goal is to identify all metrics with a significant correlation to defective samples.

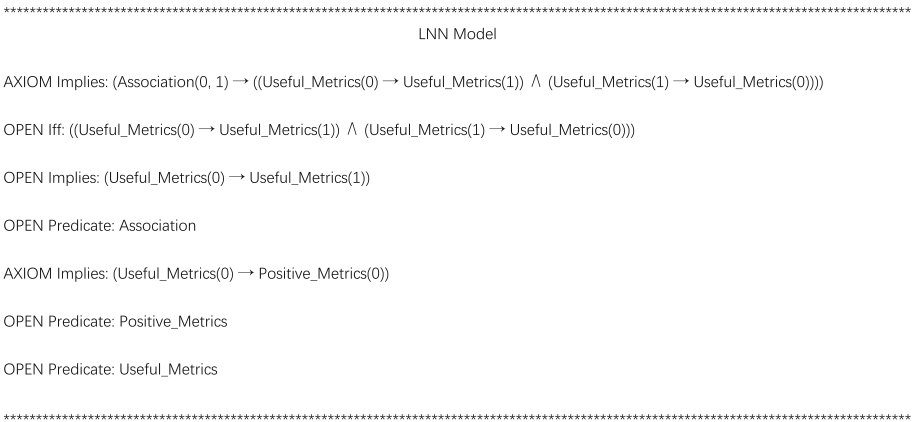


Fig. 4. The Structure of LNN Model

The reasoning process is more clearly depicted in Fig. 5, where “ \oplus ” represents “or,” and “ \otimes ” represents “and”. Rule 1, Rule 2, and Rule 3 constitute very simple logical propositions, each characterized by one single antecedent. This implies that the metric present in the antecedent of these rules must be considered as *Useful Metrics*. Since only one “antecedent” follows “IF”, it indicates that this singular metric will determine the target’s classification as a defective sample or not. Based on Rule 1 and Rule 2, it can be easily inferred that either LOC COMMENTS (metric1) or HALSTEAD ERROR EST (metric2) is associated with the defective sample. Additionally, CYCLOMATIC COMPLEXITY (metric3) is associated with negative samples. Based on Rules 4 and 5, HALSTEAD ERROR EST (metric2), CYCLOMATIC COMPLEXITY (metric3), and NUM UNIQUE OPERATORS (metric4) appear in multiple rules, suggesting an association relationship among them. We assume that metrics associated

with positive metrics are highly likely to be positive themselves. In conclusion, given the association between code metrics and Rule 6, combined with the negative metric of CYCLOMATIC COMPLEXITY (metric3), it is inferred that NUM UNIQUE OPERATORS (metric4) possesses a positive metric. Since if both CYCLOMATIC COMPLEXITY (metric3) and NUM UNIQUE OPERATORS (metric4) are negative metrics, Rule 6 cannot have a label of 1, thus NUM UNIQUE OPERATORS (metric4) must be a positive metric. In summary, the reasoning process classifies metrics into three categories: *Positive Metrics* represents metrics directly identified as defective samples, *Negative Metrics* encompasses metrics directly identified as non-defective samples, and *Useful Metrics* includes other metrics that appear in the rules. Metrics that appear multiple times across different rules are designated as *Association*. We consider metrics associated with positive metrics to be positive as well, thus identifying all positive metrics within the set.

Subsequently, only these positive metrics are utilized as attributes for prediction in the deep neural network. In this way, the LRNN layer and the neural network classifier are integrated together. The above is a simple illustration of the reasoning process using an example; in fact, most of the rules are long and complex, such as Rule 7 and Rule 8. Therefore, the entire reasoning process utilizes the Python API for LNN developed by Riegel et al. [26].

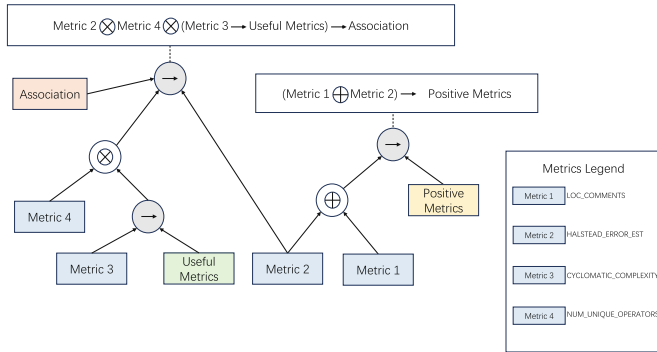


Fig. 5. Process of Reasoning

4 Experiments

In this section, we present some experiments to verify the effectiveness of the LRNN.

4.1 Datasets

We chose open-source projects in diverse programming languages from the NASA [28], PROMISE [2], AEEEM [7], widely used for defect prediction research. Details of the code repositories are in Table 3.

The total dataset is equally and randomly split into two sections (formal logic rule set and learning set). We use an formal logic set to generate rules, and for training, validation, and testing the model, a learning set is utilized. Using the NASA dataset as an example, the ratio of defective to non-defective samples is illustrated in the Fig. 6. The other two datasets, PROMISE and AEEEM, were similarly divided to generate formal logic rules specific to each dataset.

The learning set is used to construct predictive models. It is randomly divided into 70% for the training set and 30% for the testing set. The training set is used to train the parameters for deep learning, while the test set is used to evaluate the performance of LRNN and other baseline models. Besides conducting experiments on the test set, we also separately carried out experiments on each project within the three datasets. For example, using the NASA dataset, we evaluated these classifiers separately on CM1, JM1, KC1, PC1, and other projects.

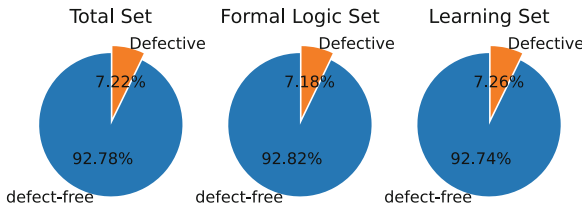


Fig. 6. Partitioning of the NASA Dataset

4.2 Baseline Methodology

We applied the prevailing algorithms in software defect prediction (DP) as baseline methods and benchmarked them against our proposed LRNN model, ensuring identical conditions for both environment and data processing. The following technologies comprise networks specifically designed to address the SDP task.

Additionally, we explored making predictions through rule learning, whereby predictions are directly based on formal logic rules. Furthermore, a neural network employing PCA for dimensionality reduction is utilized to compare the data dimensionality reduction capabilities of the LRNN.

Recently, large language models (LLMs) have demonstrated significant performance improvements in the SDP task [29]. However, to maintain the consistency of the baseline, neural network classifiers were selected for comparison.

Table 3. Details of Projects in Multiple Datasets

Dataset	Project	Number of metrics	Number of total instances	Number of defective instances	Percentage of defective instances
NASA	CM1	22	498	49	9.84
	JM1	22	10885	8779	80.65
	KC1	22	2109	326	15.46
	KC2	22	522	105	20.11
	KC3	40	458	43	9.39
	KC4	40	125	61	48.80
	MC1	39	9466	68	0.72
	MC2	40	161	52	32.30
	MW1	40	403	61	15.14
	PC1	40	1107	76	6.87
	PC2	40	5589	23	0.41
	PC3	40	1563	160	10.24
	PC4	40	1458	178	12.21
	PC5	39	17186	516	3.00
PROMISE	ant-1.7	21	745	166	22.30
	ivy-2.0	21	352	40	11.40
	camel-1.6	21	965	188	19.50
	jedit-4.0	21	306	75	24.50
	log4j-1.2	21	109	37	33.90
	Lucene-2.4	21	195	91	46.70
	poi-2.0	21	314	37	11.80
	Synapse-1.1	21	222	60	27.00
	velocity-1.6	21	229	78	34.10
	Xerces-1.3	21	453	60	15.20
	tomcat	21	858	77	8.90
Xalan-2.4	21	723	110	15.20	
AEEEM	EQ	62	324	129	39.81
	JDT	62	997	206	20.66
	LC	62	691	64	9.26
	ML	62	1862	245	13.16
	PDE	62	1497	209	13.96

1. DP-LSTM: A bidirectional long short-term memory network is utilized to obtain semantic features of the code for defect prediction [6].
2. DP-CNN: The DP-CNN leverages deep learning to generate features effectively [20].

3. DP-Transformer: The DP-Transformer captures both syntactic and semantic features from programs, utilizing these characteristics to enhance defect prediction [32].
4. Rule Learning: Predictions on the test set are made directly with the generated rules [15].
5. PCA-NN: Before the deep neural network training, the principal component analysis (PCA) algorithm was implemented for feature processing [1].

4.3 Evaluation Metrics

To comprehensively evaluate the effectiveness of the LRNN model, we use well-established metrics such as Precision (P), Recall (R), Accuracy (A), and F-measure (F) [27]. Differences between Precision, Recall, and Accuracy are outlined in [30]. These metrics are frequently used in defect prediction to assess the performance of a model.

Table 4. Confusion Matrix

	Predicted defective	Predicted non-defective
Actual defective	TP	FN
Actual non-defective	FP	TN

Prediction performance evaluation often involves examining data presented in a confusion matrix. This matrix reports the classification outcomes of a prediction model across various defect categories, compared to their actual classifications. Table 4 displays a confusion matrix containing four results for defect prediction. Here, the confusion matrix categories-true positive (TP), false negative (FN), false positive (FP), and true negative (TN)-represent the count of defective instances correctly predicted as defective, defective instances incorrectly predicted as non-defective, non-defective instances incorrectly predicted as defective, and non-defective instances correctly predicted as non-defective, respectively. With the confusion matrix, the following performance evaluation measures, commonly used in defect prediction studies, can be defined as shown in the subsequent formulas:

$$P = \frac{TP}{TP + FP}, R = \frac{TP}{TP + FN}, A = \frac{TP + TN}{TP + FN + FP + TN} \quad (5)$$

$$F\text{-Measure} = 2 \times \frac{P \times R}{P + R} \quad (6)$$

It is important to note that metrics Recall (R) and Accuracy (A) are basically unaffected by class imbalances, suggesting that both metrics perform well as long as the model is valid. Conversely, metrics Precision (P) and F-measure (F) may exhibit significantly lower prediction.

This phenomenon can be attributed to the imbalance in datasets, which often contain an excessive number of defect-free instances. The deep learning model learns the characteristics of this kind of data, thereby facilitating the accurate classification of defect-free instances (negative samples). This leads to significantly high true negative (TN) and markedly low false negative (FN). According to Eq. (5), it can be concluded that the sample imbalance problem biases both Recall (R) and Accuracy (A) towards higher levels, thereby diminishing their utility for performance evaluation.

However, in software defect prediction, the primary concern is accurately identifying whether samples are defective (positive samples). Defect-free samples are not the focus of our study. Precision (P) measures the correctness of positive sample predictions. According to Eq. (6), F-measure (F), which is the average of Precision (P) and Recall (R), is also directly influenced by Precision (P). Consequently, to a certain extent, F-measure (F) can reflect the correctness of classifying defective samples as well, a key concern in addressing the issue of class imbalance.

Therefore, the criterion for determining whether class imbalance has been resolved involves assessing whether metrics such as Recall (R) and Accuracy (A) remain high, while Precision (P) and F-measure (F) demonstrate significant improvement.

4.4 Results

The efficacy of our proposed approach is verified through comparison with existing leading deep learning algorithms in the field of software defect prediction. The experiment results are shown in Table 5, and Figs. 7, 8 and 9. We conducted a comprehensive assessment of three code repositories, along with individual evaluations for each specific project contained within these repositories.

Table 5 outline the numerical distribution for both the LRNN and baseline models across the entire three datasets. Each column in the tables represents a score for one of the performance metrics, with the best scores highlighted in bold. It is evident that each algorithm achieves high accuracy (A). However, as discussed in Sect. 4.3, we should focus more on precision (P). A closer examination of the first column reveals that LRNN scores significantly higher than the other models while also maintaining acceptable accuracy (A). Among all methods, LRNN achieves the highest precision (P) and F-measure(F). This indicates that the LRNN model effectively mitigates the impact of class imbalance. Also, since the scores are based on the entire dataset, we consider LRNN to have the ability to generalize.

In Figs. 7, 8 and 9, box plots were used to represent the specific scores of each project in the code repositories. Simply, it represents the distribution of performance metrics for each project. The red horizontal line in the graph represents the average score of the projects. A higher red line indicates that this model performs well overall on multiple projects. The red dots represent outliers, indicating that there is a significant discrepancy in the experimental results across different code projects. More red dots indicate that the model is less robust. In other words,

the same predictive model exhibits different capabilities for different project data. Of the four small graphs produced for each dataset, the two on the left represent metrics highly affected by class imbalance. The two on the right represent metrics less affected by class imbalance. We therefore focus on the two graphs on the left to determine if the LRNN scores are significantly higher than those of the other models. According to the figure, the precision (P) and F-measure (F) metrics of LRNN are significantly higher than those of the other models, while the accuracy (A) and Recall (R) metrics are not significantly different. Additionally, the prediction results of the PCA-based dimensionality reduction method show a large disparity across projects, whereas LRNN does not exhibit this problem. The figures illustrate that the LRNN model not only demonstrates robustness but also outperforms other models in terms of performance.

In conclusion, compared to other models, LRNN has achieved excellent scores across various projects, indicating its potential for broader application. Crucially, it mitigates the impact of class imbalance, demonstrating both high performance and robustness in different projects.

Table 5. Results on Multiple Datasets

Dataset	Method	Precision	Recall	F measure	Accuracy
NASA	DP-LSTM	0.5978	0.7813	0.6096	0.7996
	DP-CNN	0.5925	0.7788	0.5983	0.7847
	DP-Transformer	0.6029	0.8120	0.6123	0.8120
	Rule Learning	0.6011	0.7572	0.6210	0.8255
	PCA-NN	0.6025	0.7805	0.6195	0.8134
	LRNN	0.8039	0.8010	0.8006	0.8010
PROMISE	DP-LSTM	0.6609	0.6954	0.6731	0.7775
	DP-CNN	0.6414	0.6791	0.6528	0.7575
	DP-Transformer	0.6466	0.6884	0.6589	0.7595
	Rule Learning	0.6537	0.6693	0.6604	0.7815
	PCA-NN	0.6792	0.7066	0.6902	0.7955
	LRNN	0.7348	0.7321	0.7313	0.7321
AEEEM	DP-LSTM	0.6621	0.7039	0.6764	0.7881
	DP-CNN	0.6619	0.7101	0.6772	0.7843
	DP-Transformer	0.6742	0.7179	0.6896	0.7973
	Rule Learning	0.6751	0.6766	0.6758	0.8122
	PCA-NN	0.6907	0.7165	0.7017	0.8159
	LRNN	0.7214	0.7209	0.7207	0.7209

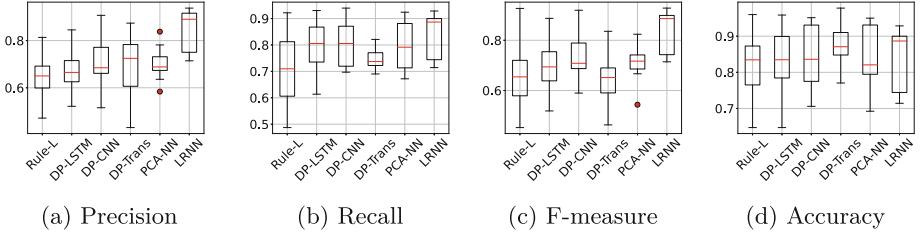


Fig. 7. On the NASA Dataset

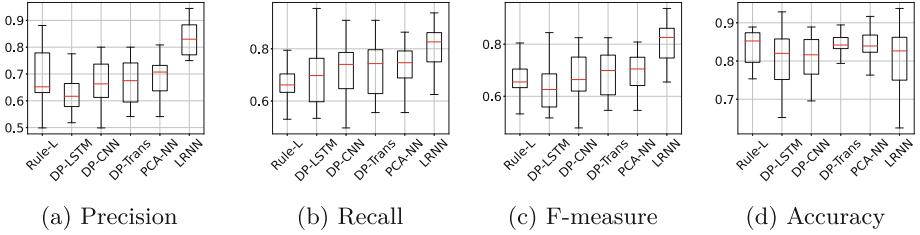


Fig. 8. On the PROMISE Dataset

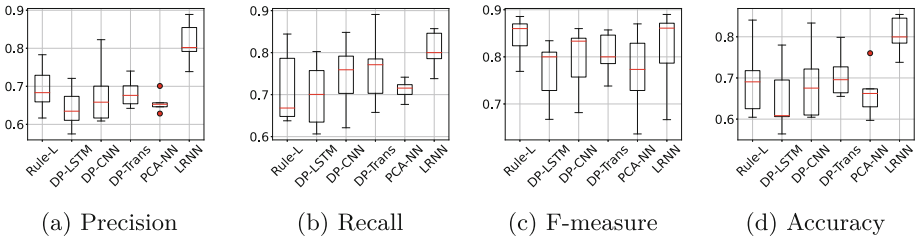


Fig. 9. On the AEEEM Dataset

4.5 Discussion

This paper presents a novel algorithm that establishes a new benchmark in the NASA, PROMISE, and AEEEM repositories, surpassing other models such as the rule learning-based method, DP-LSTM, DP-CNN, DP-Transformer, and DNN with PCA in performance. In contrast to traditional rule learning methods, LRNN transforms the generated rules into a penalty term for the attributes of the neural network, rather than making predictions based on the rules directly. It utilizes the efficient computational power of neural networks, ultimately improving prediction correctness. Compared to LSTM, CNN, and Transformer-based architectures, LRNN, based on LNN, is a model that can reason about the relationships between different features and perform feature selection. It provides a new method for data dimensionality reduction. In contrast to PCA, LRNN is a supervised learning method, not unsupervised, and possesses generalizability and robustness across different datasets.

There is substantial potential for improving the performance of software defect prediction models. LRNN requires sampling a subset of the project to develop formal logic rules, which are then utilized for feature selection to enhance the prediction accuracy. However, the applicability of this formal logic to other projects, specifically for cross-project forecasting, remains uncertain. This is because logic rules are often derived from analyzing data specific to a particular project. It's crucial to emphasize the extensive array of GitHub repositories accessible in the real world, which present considerable opportunities for advancing deep learning in software defect prediction. Cross-Project defect prediction is a notable research area, well-suited for integrating logic rules with within-project forecasting methods to enhance model generalizability. In addition to addressing software defect prediction as a binary classification method, the approach can be extended to handle multi-class classification problems in the future.

4.6 Threats to Validity

The following points highlight various potential risks to the validity of our experiments:

- Bias of dataset. A single prediction model may produce varying results across different code projects, indicating that consistent performance cannot be guaranteed in diverse project environments. In evaluating the LRNN model, we have selected multiple projects from three code repositories: NASA, PROMISE, and AEEEM. Analyses of multiple code projects have led to similar conclusions.
- Bias of evaluation measures. The potential bias stems from employing metrics like Recall and Accuracy to report on defect prediction performance, while alternative metrics like F-measure is not considered. Both Precision and F-measure are comprehensive metrics that could offer a more nuanced understanding of model performance. In our study, we employ widely used metrics such as Precision, Recall, Accuracy and F-measure for the empirical assessment of defect prediction.
- Comparison accuracy. Many authors of the related work being compared do not make the program codes of their methods available. We have meticulously implemented these methods based on the detailed descriptions provided in their respective papers.

5 Conclusion

This paper introduces a new system named Logic Rules Neural Network (LRNN) for detecting defects in software. It achieves the highest performance across all major related models on the NASA, PROMISE, AEEEM datasets. The experiment results demonstrate that LRNN can infer code metrics related to defective samples based on first-order logic rules. The impact of class imbalance is reduced without sampling. Employing these code metrics as model attributes introduces a novel method for feature engineering. Like principal components analysis (PCA), this method makes the data dimension reduction while offering interpretability.

References

1. Abdi, H., Williams, L.J.: Principal component analysis. *Wiley Interdisciplinary Rev. Comput. Stat.* **2**(4), 433–459 (2010)
2. Boetticher, G.: The promise repository of empirical software engineering data (2007). <http://promisedata.org/repository>
3. Cohen, W.W.: Fast effective rule induction. In: *Machine Learning Proceedings 1995*, pp. 115–123. Elsevier (1995)
4. Cropper, A., Dumančić, S.: Inductive logic programming at 30: a new introduction. *J. Artif. Intell. Res.* **74**, 765–850 (2022)
5. Dash, T., Chitlangia, S., Ahuja, A., Srinivasan, A.: A review of some techniques for inclusion of domain-knowledge into deep neural networks. *Sci. Rep.* **12**(1), 1040 (2022)
6. Deng, J., Lu, L., Qiu, S.: Software defect prediction via lstm. *IET Softw.* **14**(4), 443–450 (2020)
7. D’Ambros, M., Lanza, M., Robbes, R.: Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empir. Softw. Eng.* **17**, 531–577 (2012)
8. Esteva, F., Godo, L.: Monoidal t-norm based logic: towards a logic for left-continuous t-norms. *Fuzzy Sets Syst.* **124**(3), 271–288 (2001)
9. Fagin, R., Riegel, R., Gray, A.: Foundations of reasoning with uncertainty via real-valued logics. arXiv preprint [arXiv:2008.02429](https://arxiv.org/abs/2008.02429) (2020)
10. Fürnkranz, J.: A pathology of bottom-up hill-climbing in inductive rule learning. In: Cesa-Bianchi, N., Numao, M., Reischuk, R. (eds.) *ALT 2002. LNCS (LNAI)*, vol. 2533, pp. 263–277. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36169-3_22
11. Fürnkranz, J., Gamberger, D., Lavrač, N.: *Foundations of rule learning*. Springer Science & Business Media (2012)
12. Fürnkranz, J., Kliegr, T.: A brief overview of rule learning. In: Bassiliades, N., Gotlob, G., Sadri, F., Paschke, A., Roman, D. (eds.) *RuleML 2015. LNCS*, vol. 9202, pp. 54–69. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21542-6_4
13. Fürnkranz, J., Widmer, G.: Incremental reduced error pruning. In: *Machine Learning Proceedings 1994*, pp. 70–77. Elsevier (1994)
14. Gudyś, A., Sikora, M., Wróbel, L.: Rulekit: a comprehensive suite for rule-based learning. *Knowl.-Based Syst.* **194**, 105480 (2020)
15. Gudyś, A., Sikora, M., Wróbel, L.: Separate and conquer heuristic allows robust mining of contrast sets in classification, regression, and survival data. *Expert Systems with Applications*, p. 123376 (2024)
16. Guidotti, R., Monreale, A., Ruggieri, S., Turini, F., Giannotti, F., Pedreschi, D.: A survey of methods for explaining black box models. *ACM Comput. Surv. (CSUR)* **51**(5), 1–42 (2018)
17. Hitzler, P., Sarker, M.K.: Neuro-symbolic artificial intelligence: The state of the art (2022)
18. Hu, Z., Ma, X., Liu, Z., Hovy, E., Xing, E.: Harnessing deep neural networks with logic rules. arXiv preprint [arXiv:1603.06318](https://arxiv.org/abs/1603.06318) (2016)
19. Jiang, Y., Cukic, B., Ma, Y.: Techniques for evaluating fault prediction models. *Empir. Softw. Eng.* **13**, 561–595 (2008)
20. Li, J., He, P., Zhu, J., Lyu, M.R.: Software defect prediction via convolutional neural network. In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 318–328. IEEE (2017)

21. Li, T., Srikumar, V.: Augmenting neural networks with first-order logic. arXiv preprint [arXiv:1906.06298](https://arxiv.org/abs/1906.06298) (2019)
22. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.* **33**(1), 2–13 (2006)
23. Muggleton, S., De Raedt, L.: Inductive logic programming: theory and methods. *J. Logic Programm.* **19**, 629–679 (1994)
24. Pelayo, L., Dick, S.: Evaluating stratification alternatives to improve software defect prediction. *IEEE Trans. Reliab.* **61**(2), 516–525 (2012)
25. Qu, M., Tang, J.: Probabilistic logic neural networks for reasoning. *Advances in neural information processing systems* **32** (2019)
26. Riegel, R., et al.: Logical neural networks. arXiv preprint [arXiv:2006.13155](https://arxiv.org/abs/2006.13155) (2020)
27. Sasaki, Y., et al.: The truth of the f-measure. *Teach Tutor Mater* **1**(5), 1–5 (2007)
28. Shepperd, M., Song, Q., Sun, Z., Mair, C.: Data quality: some comments on the nasa software defect datasets. *IEEE Trans. Software Eng.* **39**(9), 1208–1215 (2013)
29. Song, W., Gan, L., Bao, T.: Software defect prediction via code language models. In: 2023 3rd International Conference on Communication Technology and Information Technology (ICCTIT), pp. 97–102. IEEE (2023)
30. Streiner, D.L., Norman, G.R.: “precision” and “accuracy”: two terms that are neither. *J. Clin. Epidemiol.* **59**(4), 327–330 (2006)
31. Xu, Z., et al.: Software defect prediction based on kernel pca and weighted extreme learning machine. *Inf. Softw. Technol.* **106**, 182–200 (2019)
32. Zhang, Q., Wu, B.: Software defect prediction via transformer. In: 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), vol. 1, pp. 874–879. IEEE (2020)
33. Zhou, Z.H.: Machine learning. Springer Nature (2021)



Quantitative Symbolic Robustness Verification for Quantized Neural Networks

Mara Downing¹(✉), William Eiers², Erin DeLong¹, Anushka Lodha¹, Brian Ozawa Burns¹, Ismet Burak Kadron¹, and Tevfik Bultan¹

¹ University of California Santa Barbara, Santa Barbara, USA
{maradowning, erindelongs, anushkalodha, brianozawaburns, kadron, bultan}@ucsb.edu

² Stevens Institute of Technology, Hoboken, USA
weiers@stevens.edu

Abstract. With the growing prevalence of neural networks in computer systems, addressing their dependability has become a critical verification problem. In this paper, we focus on quantitative robustness verification, i.e., whether small changes to the input of a neural network can change its output. In particular, we perform quantitative symbolic analysis, where the goal is to identify how many inputs in a given neighborhood are misclassified. We target quantized neural networks, where all values in the neural network are rounded to fixed point values with limited precision. We employ symbolic execution and model counting to achieve quantitative verification of user-defined robustness properties where the verifier will report not only whether the robustness properties are satisfied for the given neural network, but also how many inputs violate them. This measure enables comparison of non-robust networks by assessing the level of robustness, which is not possible with existing quantized network verifiers. We implement and evaluate our approach as a tool called *VerQ²*. To the best of our knowledge, *VerQ²* is the first quantitative verifier for quantized neural networks.

Keywords: Formal Verification · Model Counting · Neural Networks

1 Introduction

Neural networks are becoming increasingly common in safety critical domains like medicine and automotive industry. For safety critical applications, traditional techniques for evaluating effectiveness of neural networks, such as accuracy on a previously unseen data set, are not sufficient. It is necessary to evaluate the dependability of neural networks—for example, checking if (potentially adversarial) small changes to the input can change the output of a network, and, furthermore determining how many inputs exist within a perturbation region that can trigger an unexpected output.¹

¹ This research is supported by the NSF under Awards #2124039 and #2008660.

A full precision neural network uses floating point values, which is computationally intensive and may not be feasible in cases where storage space or processing power is limited [5]. Quantized networks address this limitation by using fixed point numbers. These networks can thus be implemented using less storage, and can also be computed faster [22]. The most extreme example of this type of network is a binary neural network, where each value is 1 or 0. Quantized networks are commonly used in mobile, embedded, and IoT devices where memory and power are limited [22].

In the safety critical healthcare domain, the use of machine learning models is expanding [28] including systems that are expected to make and execute a decision without physician involvement, such as a device that is implanted into a patient’s body. The FDA maintains strict requirements for device approval [1]. Additionally, implanted devices have strict power and size requirements which make quantized networks a valuable method of storing and executing neural networks in the medical domain.

Thus, automated verification of quantized neural networks is a critically important area of research. In this paper, we focus particularly on quantitative verification of quantized networks. Given a correctly classified input and an allowed perturbation around that input, traditional robustness verification evaluates whether or not any inputs exist in the region that are classified differently (incorrectly). Quantitative robustness verification goes further to count how many inputs in that region are classified differently. As we mention in our related work discussion, there are traditional verifiers for floating point networks and quantized networks, and there are quantitative verifiers for floating point networks and binary precision networks. However, to the best of our knowledge, this paper is the first to address quantitative robustness for quantized networks with greater than binary precision.

In this paper, we present a quantitative verifier for quantized neural networks, capable of handling different levels of precision. Our major contributions, implemented in our tool *VerQ²* (VERifier for Quantitative robustness of Quantized neural networks), are:

- A quantitative robustness verification approach for quantized neural networks based on symbolic execution and model counting.
- Improvements to constraint solving during symbolic execution for neural networks based on 1) Abstract symbolic execution, and 2) Model generation at symbolic execution tree nodes.
- Translation rules from fixed point arithmetic computations to integer constraints.
- Experimental evaluation of *VerQ²* and empirical comparison of model counters on constraints generated by neural networks.

We test our quantitative robustness verification tool *VerQ²* on networks trained from two medical datasets from the UCI Machine Learning Repository [16]. Our experiments show the techniques we present improve the performance of quantitative symbolic execution over our baseline implementation and perform better than sampling-based approaches including PROVERO [8].

Motivating Example. Let us look at two networks, both trained with the Parkinson’s dataset [16, 27] to detect Parkinson’s disease from voice data. The networks take in inputs with 22 input features, each one corresponding to a different measurement gathered from the voice of a patient. Network A has one hidden layer of size 60, and an accuracy of 84.21%. Network B has two hidden layers of size 15, and an accuracy of 89.47%. We evaluate robustness of these two networks for a robustness region using data generated from a patient who has Parkinson’s disease, where two of the 22 input features are perturbed and can take on any possible value within the expected input range.

Within the given perturbation region, both networks classify some inputs incorrectly. For a traditional verifier, this would be the extent of the robustness check results. However, with our quantitative verifier $VerQ^2$, we can further show that out of 1089 possible inputs in the perturbation region (33 allowed values per perturbed input feature), network A misclassifies only 12, whereas network B misclassifies 273. We can alternately analyze an average of multiple robustness regions using the quantitative robustness definition we provide in this paper (Sect. 2.1). Our quantitative robustness definition provides a robustness value between 0 and 1 (1 corresponds to 100% robustness—no input in the robustness region violates the robustness property). When we compare 11 robustness regions for networks A and B using $VerQ^2$, we find network A has an average robustness of 0.904, whereas network B has an average robustness of 0.954. Depending on the importance of an individual input, we may decide either to prioritize overall higher robustness or higher robustness in a key instance when choosing which network to use. Finally, we can also compare the least robust input from a set of inputs—network A has an input with 0.427 robustness whereas for network B, for the set of inputs we analyzed, the minimum robustness is 0.749. Quantitative robustness analysis enables us to make these types of comparisons among networks, which are not possible with traditional robustness analysis.

Related Work. There is a significant amount of prior work on non-quantitative verification of full precision neural networks [13, 14, 17, 18, 26, 31] and there is a limited amount of prior work on quantitative verification of full precision neural networks [8, 12, 21, 29]. These employ a variety of techniques, including formal verification, sampling, and abstraction.

One of the techniques used for scalability in symbolic verification of full-precision networks is the approximation of floating point computations using real values. However, this approach can lead to incorrect verification results [25]. Our approach avoids this type of erroneous analysis by avoiding real approximations of the quantized values and taking into account how the values will behave with rounding (Sect. 3.2).

For quantitative verification of full precision networks, one approach is to use statistical sampling methods to obtain a probabilistically sound result for quantitative robustness [8, 12]. We choose one such approach [8] for experimental comparison as it uses higher precision in published analysis.

There is also some prior work on verification methods for binarized neural networks [3, 9, 24, 36]. These have a computationally simpler task compared to higher precision quantized network verifiers such as ours.

To the best of our knowledge, there is no prior work on quantitative verification for quantized networks. In terms of traditional (non-quantitative) verification of quantized networks, there are some verifiers that use SMT constraint solving [20, 23]. However, as these are traditional verifiers, the information they can provide is limited. We discuss this drawback in Sect. 2 and show how our quantitative verification approach offers more valuable results.

As *VerQ*² is the first tool to produce quantitative robustness results for quantized networks, experimental comparison with existing tools is limited. Our approach is faster and can handle larger networks than [21], which computes exact counts for constraints on floating point networks. For comparison, we implement an algorithm proposed for quantitative verification of full precision networks [8] (with modifications for quantized networks), and show that our approach performs better in Sect. 4.

2 Quantitative Robustness Formalization

We use \mathbb{F} to denote the set of all floating point numbers, and \mathbb{I} to denote the set of fixed point numbers, with subscripts q and d indicating the total bit length and the number of fractional bits, e.g., $\mathbb{I}_{q,d}$.

Figure 1 shows a small example network where weights are marked along their respective arrows, biases are marked above respective neurons. A neural network \mathcal{N} takes an input X consisting of N features, $\langle x_0, \dots, x_{N-1} \rangle$, has K hidden layers where each hidden layer k with size J_k consists of neurons $h_{0,k}, \dots, h_{(J_k-1),k}$,

and returns values of J output neurons y_0, \dots, y_{J-1} . In a full precision network, $x_0, \dots, x_{N-1} \in \mathbb{F}$, $y_0, \dots, y_{J-1} \in \mathbb{F}$, and all $h \in \mathbb{F}$. For computing the values of the hidden and output neurons, the connections between each neuron in different layers have weights $w_{i,j,k}$ where the i th neuron in k th layer and j th neuron in $(k+1)$ th layer is connected, and each neuron i in layer j has bias value $b_{i,j}$. We use the *ReLU* activation function in our model: $ReLU(x) = \max(x, 0)$. For classification tasks, the output neuron with the highest value determines the class—if y_j has the highest value, the classification is class j .

The equations below describe how the values are computed in a non-quantized (using floating point numbers, \mathbb{F}) neural network:

$$h_{j,0} = ReLU(\sum_{i=0}^n w_{i,j,0} x_i + b_{j,0}) \quad (1)$$

$$h_{j,k} = ReLU(\sum_{i=0}^n w_{i,j,k} h_{i,(k-1)} + b_{j,k}) \quad (2)$$

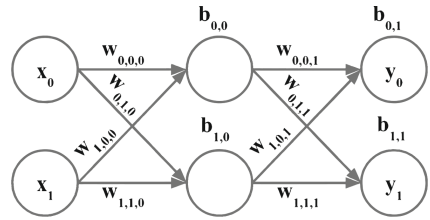


Fig. 1. A feedforward network with two input features (x_0, x_1), two outputs (y_0, y_1).

$$y_j = \sum_{i=0}^n w_{i,j,K} h_{i,K-1} + b_{j,K} \quad (3)$$

Quantization maps variables of the network (neuron values, weights, bias values, etc.) to a limited precision. We use $L_{min}, L_{max} \in \mathbb{I}$ to denote the limits of the quantized values (minimum and maximum possible values represented by the chosen fixed point size): $L_{min} = -(2^{q-d-1})$ and $L_{max} = 2^{q-d-1} - 2^{-d}$ for quantization $\mathbb{I}_{q,d}$. When working with quantized networks, the computation of nodes must not only take into account the ReLU activation functions, but also the rounding of results to maintain limited precision and avoid overflow. Due to the higher precision to avoid overflow during the computation, the ReLU piece of the above Eqs. (1) and (2) is computed as follows:

$$ReLU(expr) = \begin{cases} 0, & \text{if } expr \leq 0 \\ L_{max}, & \text{if } expr \geq L_{max} \\ expr, & \text{otherwise} \end{cases} \quad (4)$$

Additionally, Eq. (3) is replaced with the following for quantized networks:

$$y_j = \begin{cases} L_{min}, & \text{if } \sum_{i=0}^n w_{i,j,K} h_{i,K-1} + b_{j,K} \leq L_{min} \\ L_{max}, & \text{if } \sum_{i=0}^n w_{i,j,K} h_{i,K-1} + b_{j,K} \geq L_{max} \\ \sum_{i=0}^n w_{i,j,K} h_{i,K-1} + b_{j,K}, & \text{otherwise} \end{cases} \quad (5)$$

Due to the multiplication of fixed points, results of $\sum_{i=0}^n w_{i,j,k} h_{i,k-1} + b_{j,k}$ will contain double the original fractional bits. The rounding to account for this will be shown in Sect. 3.2.

2.1 Quantitative Robustness

In neural network verification, local robustness is measured by checking if any small perturbations made to the input change the output (classification result). If there exists such a perturbation that changes the output, then the network is not robust on that input. However, this yes/no answer does not give any information about how many of the perturbations change the output. For example, in Fig. 2, both examples would be determined not robust for region 2 by a traditional verifier, whereas a quantitative verifier can distinguish the levels of robustness for these two networks.

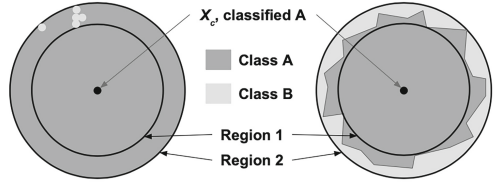


Fig. 2. Two hypothetical perturbation regions for the same input for two different networks. Both networks are robust for region 1 but not region 2. However, the network on the right is less robust than the network on the left.

We provide a definition for quantitative robustness which measures the proportion of inputs within a given perturbation region which do not change the

output. For a given neural network \mathcal{N} and a perturbation region containing input X_c within the region with known correct classification, we define the quantitative robustness measure $R(\mathcal{N}, S_{\text{PerturbRegion}})$ as follows:

$$R(\mathcal{N}, S_{\text{PerturbRegion}}) = |S_{\text{RobustSet}}|/|S_{\text{PerturbRegion}}| \quad \text{where} \\ S_{\text{RobustSet}} = \{\tilde{X} \mid \arg \max \mathcal{N}(\tilde{X}) = \arg \max \mathcal{N}(X_c) \wedge \tilde{X} \in S_{\text{PerturbRegion}}\}$$

In the definition above, $S_{\text{PerturbRegion}}$ denotes the set of all inputs within the perturbation region, and $S_{\text{RobustSet}}$ denotes the subset of those where the output of \mathcal{N} does not change. We call remaining inputs potentially adversarial inputs, and we define $S_{\text{AdversarialSet}}$ as follows: $S_{\text{AdversarialSet}} = S_{\text{PerturbRegion}} \setminus S_{\text{RobustSet}}$. This is a general form of robustness definition that can represent attacks such as one or two-pixel attacks for images, or general L_∞ ball constraints over the input [32,33].

A network with a higher number of misclassified inputs (i.e., larger $|S_{\text{AdversarialSet}}|$) in a given perturbation region is less robust (and thus more prone to adversarial attacks) than a network with fewer misclassified inputs in the same region.

3 Quantitative Symbolic Robustness Verification

In this section, we present our quantitative symbolic verification approach for quantized neural networks. The two major techniques we use for quantitative verification of neural networks are symbolic execution and model counting. Symbolic execution is a verification technique in which a program is evaluated by constructing a tree of all of the possible paths through the program, where each node in the tree corresponds to a branch condition in the program [6]. A symbolic state and path condition are recorded at each node of the tree; the symbolic state keeps track of the value of each variable at that point in the program and the path condition keeps track of all of the constraints necessary to reach that point in the program. We denote the path condition as a set of constraints C_p , comprised of clauses c_0, \dots, c_{t-1} . At a given branch, c_t is the new clause being introduced. It is possible that at any branch point, one or more of the possible paths may be infeasible, and if a path is infeasible there is no need to traverse further down. The feasibility of a path is checked with a constraint solver, which checks if the path constraints at each node are satisfiable.

The next technique that we use is model counting. A model counting constraint solver returns the number of solutions to a satisfiable constraint, or 0 if the constraint is unsatisfiable. Using symbolic execution and model counting one can determine which execution paths are more likely than others [19].

3.1 Symbolic Analysis for Quantitative Robustness

Our quantitative verifier uses symbolic execution to explore all possible paths of a neural network and model counting to compute the quantitative robustness

Algorithm 1. QUANTSYMROBUSTNESS(C_{in} , C_{out} , \mathcal{N})

- ▷ Symbolic analysis for computing quantitative robustness of a neural network
 ▷ Calls QUANTSYMEEXEC and a model counter (COUNT)

Input: C_{in} : set of all user constraints on inputs, derived from $S_{PerturbRegion}$; C_{out} : set of all user constraints on outputs, representing the robustness constraints from $S_{RobustSet}$; \mathcal{N} : the neural network under analysis

Output: Quantitative robustness metric R

```

1:  $PerturbRegionSize \leftarrow \text{COUNT}(C_{in})$                                 ▷ Computing  $|S_{PerturbRegion}|$ 
2:  $RobustSetSize \leftarrow \text{QUANTSYMEEXEC}(\mathcal{N}, C_{in}, C_{out})$           ▷ Computing  $|S_{RobustSet}|$ 
3: return  $RobustSetSize / PerturbRegionSize$                                ▷ Computing  $|S_{RobustSet}| / |S_{PerturbRegion}|$ 

```

measure. In this paper we focus only on networks with ReLU (Rectified Linear Unit) activation functions, which is a common activation function [26]. We use Z3’s linear integer arithmetic SMT solver [15] to evaluate the path conditions and determine if each branch is feasible. The fixed point values are converted into integers, and we create SMT formulas that are equivalent to the fixed point conditions they are describing (Sect. 3.2). We chose this encoding, rather than bit-vectors [10], to allow for the use of linear integer arithmetic model counters.

We use syntax C_{in} and C_{out} to represent constraints on input perturbations and expected output, respectively. We use a model counting constraint solver on C_{in} to get the total number of possible inputs within that region. Then, we use symbolic execution to capture all possible behaviors of the neural network \mathcal{N} for all inputs that satisfy C_{in} .

We show the overall quantitative verification approach in Algorithm 1, with symbolic execution detailed in Algorithm 2.

The symbolic execution process described in Algorithm 2 is as follows:

1. On line 1, a symbolic expression is created for the node value: the result of multiplying weights and adding the bias, before the ReLU is applied.
2. Lines 3–20 show the branching that occurs for all nodes except the last output node. There are three possible branches— $expr$ could be \leq the lower limit L_l (0 for ReLU activated nodes, L_{min} for output nodes), $expr$ could be \geq the upper limit L_{max} , or $expr$ could be between the two limits. In each case, the stored expression for the node is updated and a recursive call is made to continue exploring the tree given that outcome for the value of $node$. The if statement in lines 16–19 includes the addition of constraints to the path constraint which make sure $\text{EXPR}(node)$ is constrained to the proper result of rounding $expr$ (Sect. 3.2).
3. Lines 22–35 mimic 3–20 in structure, but the gathered constraint is conjoined with C_{out} and a call is made to COUNT to get a count of how many distinct input vectors (\tilde{X}) satisfy the gathered constraint.

The helper functions employed in Algorithm 2 are as follows: NEWSYMPVAR() creates a new symbolic variable name unique to the current node of the tree; NEXT($node$) returns the next node from the network, nodes are ordered by layer (from beginning to end), and from 0 to j ascending within each layer; EXPR($node$) returns the symbolic expression associated with the node, and can

Algorithm 2. QUANTSYMEXEC(\mathcal{N} , C_p , C_{out} , $node$)

▷ Quantitative symbolic execution of a neural network

▷ Called by QUANTSYMROBUSTNESS (Algorithm 1); Calls ISSAT (Algorithms 3, 5), CREATEEXPR (Algorithm 4), and COUNT; Uses helper functions NEWSYMPVAR, NEXT, EXPR, ISOUTPUT, ISLAST

Input: C_p : Current path constraint on symbolic input values; C_{out} : Constraint on output nodes; \mathcal{N} : the neural network under analysis; $node$: current node with indices j, k **Output:** Number of robust inputs within the perturbation region

```

1:  $expr \leftarrow \text{CONSTRUCTEXPR}(j, k)$ 
2: if  $\neg \text{ISLAST}(node)$  then
3:    $RobSS \leftarrow 0$ 
4:    $L_l \leftarrow 0$ 
5:   if  $\text{ISOUTPUT}(node)$  then
6:      $L_l \leftarrow L_{min}$ 
7:   end if
8:   if  $\text{ISSAT}(C_p, expr < 2^d L_l + 2^{d-1})$  then ▷  $expr_{rounded} \leq L_l$ 
9:      $\text{EXPR}(node) \leftarrow L_l$ 
10:     $RobSS \leftarrow RobSS + \text{QUANTSYMEXEC}(\mathcal{N}, C_p \wedge expr < 2^d L_l + 2^{d-1}, C_{out},$ 
     $\text{NEXT}(node))$ 
11:   end if
12:   if  $\text{ISSAT}(C_p, expr \geq 2^d L_{max} - 2^{d-1})$  then ▷  $expr_{rounded} \geq L_{max}$ 
13:      $\text{EXPR}(node) \leftarrow L_{max}$ 
14:      $RobSS \leftarrow RobSS + \text{QUANTSYMEXEC}(\mathcal{N}, C_p \wedge expr \geq 2^d L_{max} - 2^{d-1}, C_{out},$ 
     $\text{NEXT}(node))$ 
15:   end if
16:   if  $\text{ISSAT}(C_p, expr \geq 2^d L_l + 2^{d-1} \wedge expr < 2^d L_{max} - 2^{d-1})$  then ▷
     $expr_{rounded} > L_l \wedge expr_{rounded} < L_{max}$ 
17:      $\text{EXPR}(node) \leftarrow \text{NEWSYMPVAR}()$  ▷ create a new symbolic variable for  $node$ 
18:      $RobSS \leftarrow RobSS + \text{QUANTSYMEXEC}(\mathcal{N}, C_p \wedge expr \geq 2^d L_l + 2^{d-1} \wedge expr <$ 
     $2^d L_{max} - 2^{d-1} \wedge expr < 2^d \text{EXPR}(node) + 2^{d-1} \wedge expr \geq 2^d \text{EXPR}(node) - 2^{d-1},$ 
     $C_{out}, \text{NEXT}(node))$ 
19:   end if
20:   return  $RobSS$ 
21: else
22:    $RobSS \leftarrow 0$ 
23:   if  $\text{ISSAT}(C_p, expr < 2^d L_{min} + 2^{d-1})$  then ▷  $expr_{rounded} \leq L_l$ 
24:      $\text{EXPR}(node) \leftarrow L_l$ 
25:      $RobSS \leftarrow RobSS + \text{COUNT}(C_p \wedge expr < 2^d L_{min} + 2^{d-1} \wedge C_{out})$ 
26:   end if
27:   if  $\text{ISSAT}(C_p, expr \geq 2^d L_{max} - 2^{d-1})$  then ▷  $expr_{rounded} \geq L_{max}$ 
28:      $\text{EXPR}(node) \leftarrow L_{max}$ 
29:      $RobSS \leftarrow RobSS + \text{COUNT}(C_p \wedge expr \geq 2^d L_{max} - 2^{d-1} \wedge C_{out})$ 
30:   end if
31:   if  $\text{ISSAT}(C_p, expr \geq 2^d L_{min} + 2^{d-1} \wedge expr < 2^d L_{max} - 2^{d-1})$  then ▷
     $expr_{rounded} > L_l \wedge expr_{rounded} < L_{max}$ 
32:      $\text{EXPR}(node) \leftarrow \text{NEWSYMPVAR}()$  ▷ create a new symbolic variable for  $node$ 
33:      $RobSS \leftarrow RobSS + \text{COUNT}(C_p \wedge expr \geq 2^d L_l + 2^{d-1} \wedge expr < 2^d L_{max} -$ 
     $2^{d-1} \wedge expr < 2^d \text{EXPR}(node) + 2^{d-1} \wedge expr \geq 2^d \text{EXPR}(node) - 2^{d-1} \wedge C_{out})$ 
34:   end if
35:   return  $RobSS$ 
36: end if

```

be used to change the symbolic expression (this accesses and modifies the symbolic state); `ISOUTPUT(node)` returns True only if the node is an output node; `ISLAST(node)` returns True only if the node is the last node in the order; `CONSTRUCTEXPR` is used to compute the multiplication of previous layer nodes by weights and add the bias value, and is shown in Algorithm 4.

We use four different model counting tools to obtain the number of satisfying solutions to a constraint. Three of these are model counters: ABC [4], Barvinok [34], and LattE [7]. The fourth is Z3 [15], which can be used to count satisfying models by generating a satisfying model, adding the negation of the model to the constraint, and looping until the constraint is unsatisfiable. We describe the model counters in more detail in Sect. 3.4.

3.2 Fixed Point Rounding Constraints

As discussed in Sect. 2, after the multiplication of node values by weights, the result has double the fractional bits, and must be rounded. We translate the fixed-point inequality expressions (used when deciding how the ReLU impacts each internal node value) into equivalent integer expressions for symbolic execution of the network.

The fixed point values themselves are translated to integers—for example, a concrete fixed point value 0010.1100 (2.75) can be represented as a concrete integer value 00101100 (44) by eliminating the decimal point. We use this approach in transforming fixed point computations to equivalent integer constraints during symbolic execution of the network.

For the following formulas, d will represent the number of fractional bits in the chosen fixed point representation and $expr \in \mathbb{I}_{q,2d}$ will represent the value to be rounded. The rounding rule is as follows, with $\&$ representing bitwise AND operation, and \gg representing arithmetic right shift:

$$expr_{rounded} \in \mathbb{I}_{q,d} = \begin{cases} expr \gg d, & \text{if } expr \& (2^d - 1) < 2^{d-1} \\ (expr \gg d) + 1, & \text{if } expr \& (2^d - 1) \geq 2^{d-1} \end{cases} \quad (6)$$

The following four equations provide equivalences between the expression we wish to evaluate, using the rounded result of $expr$, and the equivalent expression without using the rounded result.

$$\begin{aligned} expr_{rounded} < x &\Leftrightarrow expr < 2^d x - 2^{d-1} \\ expr_{rounded} \leq x &\Leftrightarrow expr < 2^d x + 2^{d-1} \\ expr_{rounded} > x &\Leftrightarrow expr \geq 2^d x + 2^{d-1} \\ expr_{rounded} \geq x &\Leftrightarrow expr \geq 2^d x - 2^{d-1} \end{aligned} \quad (7)$$

It is also necessary during the symbolic execution of the network to use $expr_{rounded}$ as an argument in future layer computations. This is handled by setting $expr_{rounded}$ equal to a new symbolic variable n , and then using n in any place $expr_{rounded}$ would appear:

$$expr_{rounded} = n \Leftrightarrow expr < 2^d n + 2^{d-1} \wedge expr \geq 2^d n - 2^{d-1} \quad (8)$$

Algorithm 3. $\text{IS}_{\text{SAT}}^{\text{orig}}(C_p, c_t)$
 \triangleright Path constraint checking (simplest version)
 \triangleright Calls an SMT solver ($\text{SMT}_{\text{SOLVE}}$)

Input: C_p : prior path constraint; c_t : new constraint to be added to the path constraint

Output: Satisfiability of $C_p \wedge c_t$

```

1: if  $\text{SMT}_{\text{SOLVE}}(C_p \wedge c_t) = \text{SAT} \vee \text{SMT}_{\text{SOLVE}}(C_p \wedge c_t) = \text{UNKNOWN}$  then
2:   return SAT
3: else
4:   return UNSAT
5: end if

```

3.3 Constraint Solving Optimizations

The most time consuming computation during symbolic execution of neural networks is in evaluating the satisfiability of path constraints C_p (which we do using Z3 [15]). Therefore, we implement a few strategies to help optimize this computation. The basic constraint solving algorithm (before optimization) is shown in Algorithm 3, where we conservatively return the result as satisfiable if the constraint solver returns unknown. Note that this does not result in imprecision during symbolic execution since the constraint is preserved. It ensures that we do not eliminate paths that might be feasible but where the constraint solver is unable to prove a definite result at that stage of symbolic execution.

Abstract Symbolic Execution. The first optimization we have implemented is abstract symbolic execution, in which an abstract state is kept alongside the typical symbolic state. In this abstract state, each variable can have one of eight values $\{\perp, -, 0, +, -0, -+, 0+, \top\}$ indicating what is known about the variable’s sign, which are arranged in a complete lattice with the partial order \subset such that \perp is the meet of all elements of the lattice and \top is the join of all elements of the lattice. We show the Hasse diagram for this lattice in Fig. 3.

We chose this particular abstract domain due to the ReLU activation functions: the ReLU choice is determined by the sign of the input, and the ReLU determines the sign of its output. These abstract values are updated alongside the symbolic values held in the symbolic state; at any given program point, the abstract value for each variable will be an over-approximation of what is known about its symbolic value.

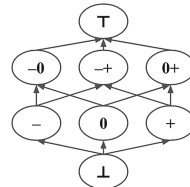


Fig. 3. Lattice for the abstract domain used in abstract symbolic execution.

Algorithm 4. CONSTRUCTEXPR(j, k)▷ Computes $\sum_{i=0}^n w_{i,j,k} h_{i,(k-1)} + b_{j,k}$

▷ Uses helper function ABSVAL to access and modify the abstract values associated with nodes and concrete values, and ABSEVAL to compute an abstract mathematical operation on two abstract values.

Input: j : index of node within layer; k : index of layer

```

1:  $expr \leftarrow \text{MULTIPLY}(\text{EXPR}(h_{0,k-1}), w_{0,j,k})$ 
2:  $\text{ABSVAL}(expr) \leftarrow \text{ABSEVAL}(\text{ABSVAL}(h_{0,k-1}) \times \text{ABSVAL}(w_{0,j,k}))$            ▷ Abstract
3: for  $i$  in range(1, $n$ ) do
4:    $expr \leftarrow \text{ADD}(expr, \text{MULTIPLY}(\text{EXPR}(h_{i,k-1}), w_{i,j,k}))$ 
5:    $\text{ABSVAL}(expr) \leftarrow \text{ABSEVAL}(\text{ABSVAL}(expr) +$ 
6:      $\text{ABSEVAL}(\text{ABSVAL}(h_{i,k-1}) \times \text{ABSVAL}(w_{i,j,k})))$            ▷ Abstract
7: end for
8:  $expr \leftarrow \text{ADD}(expr, b_{j,k})$ 
9:  $\text{ABSVAL}(expr) \leftarrow \text{ABSEVAL}(\text{ABSVAL}(expr) + \text{ABSVAL}(b_{j,k}))$            ▷ Abstract
10: return ( $expr, \text{ABSVAL}(expr)$ )
```

Before checking any SMT formula for satisfiability, the new clause c_t is checked abstractly using its abstract counterpart $c_{t,a}$, and if $c_{t,a}$ is unsatisfiable, then the branch is infeasible and there is no need to invoke Z3. If $c_{t,a}$ is a tautology, (e.g., $- < +$), $C_p \wedge c_t$ is satisfiable and c_t is not added to the path constraint. Only if $c_{t,a}$ is undetermined then Z3 is invoked on the full constraint.

The method we use to construct symbolic expressions for internal nodes can be augmented to construct these abstract values as well, as shown in Algorithm 4. The three lines we add specifically for this abstract value computation are marked with “Abstract”; without these lines, the function computes solely the symbolic values necessary for symbolic execution. The abstract satisfiability check is added as lines 1–2 and 13–14 in Algorithm 5.

Model Generation and Checking. In the symbolic execution tree, we define model m to be a model that satisfies the path constraint of the node under consideration and m_{prev} as the model that satisfies the path constraint of its parent node. When checking the satisfiability of a node’s path constraint, first we check if m_{prev} satisfies that constraint—if so, the constraint is satisfiable and m is set to be m_{prev} . If not, or if there is no m_{prev} available, we evaluate using Z3 and if it is satisfiable generate a new model m for this node. The parent model m_{prev} must satisfy one of the new clauses (together, the clauses cover the entire solution space) which marks that clause as satisfiable. Note that checking if a model satisfies a given constraint is faster to compute than checking satisfiability of a constraint. This approach uses concrete values during symbolic execution similar to concolic execution [30].

This model generation and check is added as lines 3 and 6 of Algorithm 5.

Overall Constraint Solving Algorithm. Algorithm 5 incorporates both abstract symbolic execution and model generation as discussed above.

Algorithm 5. $\text{ISAT}_{\text{final}}(C_p, c_t, c_{t,a}, m_{\text{prev}})$

▷ Path constraint checking

▷ Calls an SMT solver (SMTSOLVE) and a model generation tool (GETMODEL)

Input: C_p : set of all path constraints from prior branches, c_t : new path constraint to check, m_{prev} : model that satisfies C_p , $c_{t,a}$: equivalent abstract constraint to c_t (details of the construction of $c_{t,a}$ are provided in the appendix).

Output: Satisfiability of $C_p \wedge c_t$

```

1: if  $c_{t,a} = \text{TAUTOLOGY}$  then return SAT           ▷ Next constraint  $C_p$ , passes model  $m_{\text{prev}}$ 
2: else if  $c_{t,a} = \text{SAT}$  then
3:   if  $m_{\text{prev}} \models c_t$  then return SAT         ▷ Next constraint  $C_p \wedge c_t$ , passes model  $m_{\text{prev}}$ 
4:   else
5:     if  $\text{SMTSOLVE}(C_p \wedge c_t) = \text{SAT}$  then
6:        $m \leftarrow \text{GETMODEL}(C_p \wedge c_t)$ 
7:       return SAT                               ▷ Next constraint  $C_p \wedge c_t$ , passes model  $m$ 
8:     else if  $\text{SMTSOLVE}(C_p \wedge c_t) = \text{UNKNOWN}$  then
9:       return SAT                               ▷ Next constraint  $C_p \wedge c_t$ , no model
10:    else return UNSAT
11:    end if
12:  end if
13: else return UNSAT
14: end if

```

3.4 Model Counting Approaches

We use two model counting techniques in VerQ^2 : symbolic and constraint-loop model counting.

Symbolic Model Counting. Symbolic model counters ABC [4], Barvinok [34], and LattE [7] compute the full model count for constraints without explicitly enumerating all solutions. In VerQ^2 we use the model counting constraint solver ABC for part of the model counting. ABC is an automata-based model counter which constructs a finite-state automaton characterizing the set of solutions to a constraint. We also test LattE and Barvinok, both of which compute the model count using Barvinok’s algorithm [11].

Constraint-Loop Model Counting. In this approach, a satisfiability solver is used iteratively to find the solutions to a constraint formula F by first solving F (using an SMT-solver such as Z3) to get the model m , then re-solving $F \wedge \neg m$ ($\neg m$ indicates that m cannot be a solution). This iteration continues until either no more solutions exist or the solver returns UNKNOWN or times out. In general, if $|F|$ is the model count of F , then the solver is invoked $|F|$ times. We name this approach constraint-loop model counting. This approach still uses symbolic constraint solving just like symbolic model counting, but it iteratively produces one model at a time and must run in a loop to produce a count.

Model Counting Implementation. In our quantitative verification approach, there are two places where model counting is needed. The first is before symbolic execution, to obtain a model count of the input constraints (C_{in}) so that the robustness can be calculated as a proportion of the total number of perturbed inputs. The second is at the leaves of the symbolic execution tree, where path

constraints are conjoined with C_{out} : at each leaf, we count satisfying solutions to $C_p \wedge C_{out}$.

For the first model count, the constraints are relatively small. For the second model count, the constraints are generated by symbolic execution to represent the behavior of the network, and can be large and complex, which impacts the cost of model counting. However, the initial model count can be leveraged to allow an inverse model count ($|S_{AdversarialSet}|$) to compute robustness, which for a fully or nearly fully robust network can yield small model counts from the leaves. This property can aid the performance of constraint-loop model counting. This constitutes counting solutions to $C_p \wedge \neg C_{out}$ at each leaf.

4 Experimental Evaluation and Discussion

In our experimental evaluation we investigate the following research questions:

- RQ1:** Which of four integer model counting approaches is the best choice for counting the constraints generated by neural networks?
- RQ2:** Do the optimizations we propose improve the symbolic verification time for neural networks?
- RQ3:** Does quantitative symbolic robustness verification of neural networks produce results faster than brute force testing?
- RQ4:** Does the quantitative symbolic robustness verification approach we propose in this paper and implement in *VerQ²* perform better than the existing tool PROVERO [8]?

We trained neural networks using Tensorflow [2] on three different datasets described below, all obtained from the UCI Machine Learning Repository [16]. The networks are trained with full precision, and then converted to fixed-point. All tests use values in $\mathbb{I}_{8,4}$.

Dataset Specifics: *The Iris dataset* [16] contains four real-valued input variables (normalized to the $[0,1]$ range), and three output classifications. We use this smaller dataset for comparison to explain our choice of model counting strategies, without the need to run slower model counters on large networks for comparison. *The Parkinson’s dataset* [16,27] contains 22 real-valued input variables (normalized to the $[-1,1]$ range) and two output classifications. *The Wisconsin Breast Cancer dataset: abbrev. Cancer* [16,35] contains 30 real-valued input variables (normalized to the $[-1,1]$ range) and two output classifications. Accuracies of all of the tested networks are presented in our code repository.²

VerQ² Output: Our tool has two ways in which it may produce a robustness result—it will either report the robustness R as an exact result or as a sound upper bound. To demonstrate the output produced by *VerQ²*, we present a few examples in Table 1, all from the same network.

² All of the data from the experiments in this paper is available at <https://github.com/mara-downing/ver-q2>.

Table 1. Quantitative robustness results for different inputs computed by $VerQ^2$ for a network trained from the Parkinson’s dataset with 2 hidden layers, size 15.

$ S_{PerturbRegion} $	$ S_{AdversarialSet} $	Exact	R	Explanation
3,373,232,128	0	Yes	1	The network is fully robust for the region.
2,951,578,112	374	Yes	0.9999999	There are exactly 374 misclassifications in the region.
3,855,122,432	10,179	No	0.9999974	There are at least 10,179 misclassifications in the region.

Comparison of Model Counters: We first compare 2 model counting approaches for the initial count of the user’s input constraints: constraint-loop model counting via Z3, and symbolic model counting via ABC. The results are shown in Fig. 4a. These tests use ten constraints of each size from the Iris constraint set and only compare the time taken to complete the model count.

A radius of ∞ indicates that values are bounded by the range the inputs were normalized to when training the network.

These results are intuitive—the constraint-loop approach with Z3 needs to call Z3 once per model, so a larger count is slower. However, as ABC does not use this loop approach, all of these constraints can be counted quickly. For all future tests, we use ABC for the initial model count.

For the leaf counts, we also use the Iris networks and all constraints are of the form of a two input feature attack, where both input features can be any value within the normalized range. Results are shown in Fig. 4b. The results are an average of 10 tests with a 600s time bound each (ABC can exceed this since time is checked after each counting task is completed). The # E column indicates how many are exact counts. Incomplete counts are a sound upper bound

(a) Comparison of constraint-loop and symbolic model counting for input constraints of various sizes for the Iris dataset.

Constraint Parameters		Time (s)	
# inputs perturbed	radius	Z3	ABC
4	0.1	2.92	0.01
4	0.2	80.38	0.01
1	∞	0.22	0.01
2	∞	2.80	0.01

(b) Comparison of constraint-loop and symbolic model counting for the leaf counts of 2 input feature perturbations of the Iris networks.

Network Parameters			Z3		ABC	
# HL	HL Size	Accuracy	# E	Time (s)	# E	Time (s)
1	20	73.33	10	6.53	10	122.92
1	30	73.33	10	13.55	6	371.80
1	40	60.00	10	28.44	0	895.28
1	50	73.33	10	17.39	7	331.48
1	60	93.33	10	19.09	8	357.78
1	70	100.00	10	25.48	2	964.88
2	10	93.33	8	30.42	10	65.02
2	15	93.33	10	14.09	10	49.24
2	20	100.00	10	47.77	10	81.14
2	25	100.00	9	73.44	10	149.66
2	30	100.00	8	94.28	10	164.80
2	35	100.00	3	111.22	10	169.82
3	5	100.00	10	16.23	10	32.63
3	10	93.33	9	86.54	10	145.18
3	15	100.00	1	110.08	10	166.61
3	20	100.00	7	84.23	10	173.01
Average:			8.44	48.67	8.31	265.08

Fig. 4. Model counting performance comparison for input and leaf constraints.

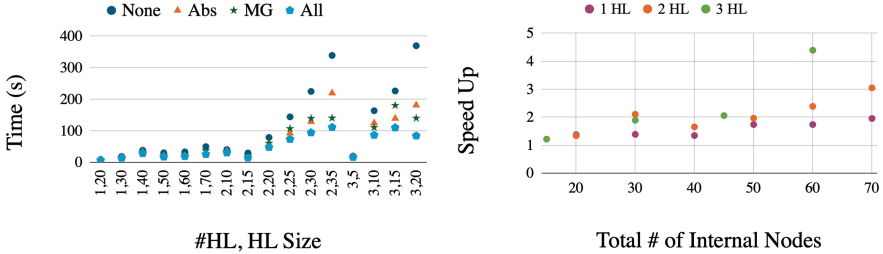


Fig. 5. Comparison of constraint solving optimization strategies on the Iris networks. Left shows times for all four levels of optimization, right shows speed up (as in *value* times speed up) caused by both optimizations.

on robustness measure R using Z3, as Z3 is counting misclassifications, and a sound lower bound on R using ABC. Z3 is able to complete all of the tests faster than ABC. Both tools report some incomplete counts. We additionally tested the leaf counts using the Barvinok and LattE model counters, but both were unable to solve simple leaf constraints in under an hour.

These experiments answer **RQ1**, identifying the best tool choice for both places where model counting is required: ABC performs better than Z3 for the initial count, and Z3 performs better than ABC for the leaf counts. All further experiments use the model counters in this configuration.

Evaluating Effectiveness of Optimization Strategies: For this set of experiments, we used the networks trained on the Iris dataset [16]. Each test uses 2 perturbed input features, each allowed to take on any value in the normalized range. Results are shown in Fig. 5, where HL stands for Hidden Layer(s). The categories are *Base*: Time with no optimizations, *Abs*: Time with abstract symbolic execution only, *MG*: Time with model generation only, and *All*: Time with all optimizations.

For all tests in Fig. 5, both abstract symbolic execution and model generation show improvement over the base solving time. The combination of the two shows an even larger improvement than either one alone. The effectiveness of these optimizations increases with a higher # of internal nodes, which is expected as more nodes means more branch points which can benefit from optimization.

We additionally tested these improvements on networks trained from the Parkinson’s dataset. We allow 3 perturbed input features, which can take on any value in the normalized range. We remove any tests where both optimized and unoptimized verification time out. These tests show an average 1.43x speedup.

Finally, we tested these improvements on networks trained on the Cancer dataset. We allow 5 perturbed input features, which can take on any value in the normalized range. Additionally, we allow one comparative perturbation—one specific input value must be greater than another. We remove any tests where both optimized and un-optimized verification time out. These tests show an average 1.13x speedup.

These results answer **RQ2**, showing that our optimizations produce improvements to symbolic verification time. Furthermore, our optimizations are more effective with increasing network size.

Comparing $VerQ^2$ with Random Sampling and Exhaustive Enumeration: In this section, we show that our approach can perform better than random sampling without replacement and exhaustive concrete enumeration.

For this section, *exhaustive concrete enumeration* refers to the approach where all valid quantized inputs within the perturbation region are tested. Meanwhile, *random sampling without replacement* (once an element has been sampled, it cannot be chosen again) functions very similarly, but is constrained by a time bound rather than a sample number and the order in which samples are taken is randomized. *Exhaustive concrete enumeration* can thus achieve an exact robustness result by taking the time to check every input, whereas *random sampling without replacement* can find a number of correctly classified and incorrectly classified samples (incorrectly classified divided by $|S_{PerturbRegion}|$ forms a sound upper bound on the robustness R).

We begin with a set of experiments comparing our approach to *exhaustive concrete enumeration*. Our results are shown in Table 2. For this table, we take 160 tests (10 tests per network size, 16 networks) and divide them into three categories by result, corresponding to the three rows of the table. The first two rows indicate that $VerQ^2$ obtained an exact robustness result for R and are split by whether or not $R = 1$. The last row indicates that $VerQ^2$ obtained a sound upper bound on R .

Table 2. Comparison of $VerQ^2$ evaluation time with Exhaustive Concrete Enumeration for 16 Parkinson’s networks, 10 constraints each.

	# Tests	$VerQ^2$ time (s)	Exhaustive Enum time (s)
$VerQ^2$ exact, $R = 1$	114	55.51	80,820.98
$VerQ^2$ exact, $R < 1$	5	73.50	90,262.19
$VerQ^2$ sound upper bound	41	1,087.67	91,522.52

Time reported for $VerQ^2$ is an average of all tests in that category, whereas time reported for *exhaustive concrete enumeration* is an extrapolation using the average time per input multiplied by the total number of inputs in the perturbation region. We used the same 16 network sizes used in Fig. 4b, trained on the Parkinson’s dataset [16, 27], 10 perturbation regions per network, and a 30 min timeout. Each constraint allows 11 input features to be perturbed with a radius of 0.2.

In Table 2 we see that in cases where $VerQ^2$ can obtain an exact result, it is 3 orders of magnitude faster than *exhaustive concrete enumeration*. To further analyze the 41 cases for which $VerQ^2$ produces a sound upper bound, we construct an additional experiment in which *random sampling without replacement*

is given the exact amount of time as $VerQ^2$ took for each given test to produce as many misclassifications as possible (if $VerQ^2$ found an exact result for a test in 2000ms, *random sampling without replacement* is given 2000ms for that test).

This random sampling strategy iteratively chooses inputs from $S_{PerturbRegion}$ at random, runs them in the network, and records whether or not they are classified as expected until the specified time limit. Results are shown in Fig. 6.

With these results, we answer **RQ3** affirmatively. Moreover, $VerQ^2$ performs better than random sampling even excluding cases where the tested region is fully robust (where $VerQ^2$ obviously outperforms random sampling).

Comparing $VerQ^2$ with PROVERO: We compare $VerQ^2$ with the sampling based tool PROVERO [8]. Given a threshold θ of proportion of adversarial (misclassified) inputs within a perturbation region, PROVERO can report with a degree of confidence measured by parameters η and δ whether or not the proportion of adversarial inputs is above or below a threshold θ . η is an additive precision on the threshold θ and δ is the level of probabilistic certainty necessary to consider a threshold proved or disproved. PROVERO is not specifically designed for our purpose, but it is an existing and effective robustness tool which does not rely on properties of floating point or binary neural networks and thus can be adapted as a baseline comparison.

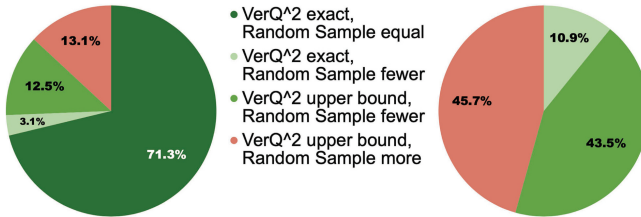


Fig. 6. Comparison with *random sampling* approach. $VerQ^2$ shows improvement in the green sections. Left: All tests; Right: All nonrobust ($R < 1$) tests.

For this comparison, we re-implement the PROVERO algorithm since the original tool does not support inequality-based perturbation region constraints and fixed-point input feature values. Additionally, since PROVERO requests an expected robustness threshold to be given by the user (θ), and our approach produces this threshold automatically, we set up a binary search loop where θ starts at 0.5 and then is modified as thresholds are proven or disproven.

We report our results for comparison with PROVERO in Table 3 and divide the rows of this table in the same way as Table 2. We report our results for the first two rows with the time taken by PROVERO and the distance between the upper and lower R discovered using the binary search. For example, if PROVERO can probabilistically prove that the robustness measure R is bounded by $0.751 \leq R < 0.764$, we report 0.013 difference between the upper and lower bound (robustness range in Table 3, averaged across all 114 or 5 cases in the row).

Table 3. Comparison of $VerQ^2$ with PROVERO, using parameters $\eta = 0.001$ and $\delta = 0.01$ for PROVERO as well as a binary search loop to find the closest probabilistically provable thresholds.

	# Tests	$VerQ^2$ time (s)	PROVERO ($\eta = 0.001, \delta = 0.01$)		
			time (s)	Avg. Rob. Range	Avg. Upper Bound Diff.
$VerQ^2$ exact, $R = 1$	114	55.51	1,800.00	0.00195	NA
$VerQ^2$ exact $R < 1$	5	73.50	1,800.00	0.00195	NA
$VerQ^2$ sound upper bound	41	1,087.67	1,715.24	NA	0.056

In Table 3, we also show the 41 cases for which $VerQ^2$ reports a sound upper bound, and display here the average difference obtained by subtracting PROVERO’s upper bound on R from our upper bound on R . For example, if PROVERO can probabilistically prove that the robustness measure R is bounded by $0.751 \leq R < 0.764$ and $VerQ^2$ reports a sound upper bound on the robustness at 0.771, we report $0.771 - 0.764 = 0.017$ as the upper bound difference in Table 3 (averaged across all 41 cases in the row).

We use δ as 0.01, the level used in [8]. We test η as 0.001, the most precise value used in [8]. For all tests, we give a 30 min timeout to match the timeout for $VerQ^2$ on these networks, and we report the results at timeout (or stop early if the robustness range becomes $\leq \eta$).

From Table 3 we can see that our approach outperforms PROVERO both by time and by precision for cases where we get an exact result—in all of these cases, we are able to produce a sound and exact result, whereas PROVERO is only able to bound the result and is producing probabilistic guarantees rather than fully sound guarantees on these bounds. For the cases where we get an upper bound, we show that our approach is faster than PROVERO as well, and additionally that, while it is expected PROVERO’s approach will produce a more precise upper bound by being able to sample and prove probabilistic results instead of counting individual models, ours is not much higher on average.

With these results, we answer **RQ4** and show that our approach performs better than PROVERO on cases where we can produce an exact result and comparable on cases where we produce a sound upper bound.

5 Discussion

Within our experiments, we evaluate different model counting approaches and find the most effective for quantized networks (**RQ1**) and we show how our constraint solving optimizations improve our quantitative verification (**RQ2**). We additionally demonstrate how our technique can perform better than two forms of brute force testing (**RQ3**) and an existing published tool [8] (**RQ4**).

It is known that worst case complexities of many symbolic verification techniques are exponential. However, worst case exponential complexity has not excluded verification techniques from transition to practice. For our approach,

we have a worst-case complexity of $O(3^{|\mathcal{N}|})$ calls to a constraint solver, where $|\mathcal{N}|$ is the number of nodes in \mathcal{N} minus the input nodes. However, despite this high worst-case complexity, within existing networks and local perturbation regions the actual calls to the constraint solver are far fewer.

6 Conclusion

We present a symbolic execution and model counting based approach for quantitative verification of quantized neural networks, and its implementation in *VerQ²*. Given a user-defined robustness property for a network, we compute the proportion of inputs in the perturbation region that do not change the output of the network, which provides a quantitative measure of robustness. We present translations from fixed-point constraints to equivalent integer constraints so that we can use integer model counting to produce quantitative results. We have compared the performance of different model counting approaches on the quantized network constraints, and also our own improvements to constraint solving within symbolic execution. Additionally, we have compared our approach against two brute-force sampling approaches and an existing tool for quantitative floating point neural network verification modified for quantized values and found our approach performs favorably against all three. To the best of our knowledge, *VerQ²* is the first quantitative verifier for quantized networks with more than binary precision.

References

1. Artificial intelligence and machine learning (ai/ml)-enabled medical devices. <https://www.fda.gov/medical-devices/software-medical-device-samd/artificial-intelligence-and-machine-learning-ai-ml-enabled-medical-devices>. Accessed 4 Dec 24
2. Abadi, M., et al.: Tensorflow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 265–283 (2016)
3. Amir, G., Wu, H., Barrett, C., Katz, G.: An smt-based approach for verifying binarized neural networks. In: Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Proceedings, Part II 27, pp. 203–222 (2021)
4. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: International Conference on Computer Aided Verification, pp. 255–272 (2015)
5. Bacchus, P., Stewart, R., Komendantskaya, E.: Accuracy, training time and hardware efficiency trade-offs for quantized neural networks on fpgas. In: International Symposium on Applied Reconfigurable Computing, pp. 121–135 (2020)
6. Baldoni, R., Coppa, E., D’elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. *ACM Comput. Surv. (CSUR)* **51**(3), 1–39 (2018)
7. Baldoni, V., et al.: A user’s guide for latte integrale v1.7.2. *Optimization* **22**(2) (2014)

8. Baluta, T., Chua, Z.L., Meel, K.S., Saxena, P.: Scalable quantitative verification for deep neural networks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 312–323 (2021)
9. Baluta, T., Shen, S., Shinde, S., Meel, K.S., Saxena, P.: Quantitative verification of neural networks and its security applications. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1249–1264 (2019)
10. Baranowski, M., He, S., Lechner, M., Nguyen, T.S., Rakamarić, Z.: An smt theory of fixed-point arithmetic. In: International Joint Conference on Automated Reasoning, pp. 13–31 (2020)
11. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.* **19**(4), 769–779 (1994), <http://www.jstor.org/stable/3690312>
12. Bu, H., Sun, M.: Measuring robustness of deep neural networks from the lens of statistical model checking. In: 2023 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2023)
13. Bunel, R., Mudigonda, P., Turkaslan, I., Torr, P., Lu, J., Kohli, P.: Branch and bound for piecewise linear neural network verification. *J. Mach. Learn. Res.* **21** (2020)
14. Chen, S., Wong, E., Kolter, J.Z., Fazlyab, M.: Deepsplit: scalable verification of deep neural networks via operator splitting. *IEEE Open J. Control Syst.* **1**, 126–140 (2022)
15. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340 (2008)
16. Dua, D., Graff, C.: Uci machine learning repository (2017). <http://archive.ics.uci.edu/ml>
17. Dvijotham, K., Stanforth, R., Goyal, S., Mann, T.A., Kohli, P.: A dual approach to scalable verification of deep networks. In: UAI, vol. 1, p. 3 (2018)
18. Gehr, T., Mirman, M., Drachler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: Ai2: safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy, pp. 3–18 (2018)
19. Geldenhuys, J., Dwyer, M.B., Visser, W.: Probabilistic symbolic execution. In: International Symposium on Software Testing and Analysis, pp. 166–176 (2012)
20. Giacobbe, M., Henzinger, T.A., Lechner, M.: How many bits does it take to quantize your neural network? In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 12079 (2020)
21. Gopinath, D., Pasareanu, C.S., Usman, M.: Quantifyml: how good is my machine learning model? In: 30th International Symposium on Software Testing and Analysis (ISSTA) (2021)
22. Guo, Y.: A survey on methods and theories of quantized neural networks. arXiv preprint [arXiv:1808.04752](https://arxiv.org/abs/1808.04752) (2018)
23. Henzinger, T.A., Lechner, M., Žikelić, Đ.: Scalable verification of quantized neural networks. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 35, pp. 3787–3795 (2021)
24. Jia, K., Rinard, M.: Efficient exact verification of binarized neural networks. *Adv. Neural. Inf. Process. Syst.* **33**, 1782–1795 (2020)
25. Jia, K., Rinard, M.: Exploiting verified neural networks via floating point numerical error. In: International Static Analysis Symposium, pp. 191–205 (2021)

26. Katz, G., et al.: The marabou framework for verification and analysis of deep neural networks. In: International Conference on Computer Aided Verification, pp. 443–452 (2019)
27. Little, M., McSharry, P., Roberts, S., Costello, D., Moroz, I.: Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection. *Nature Precedings*, p. 1 (2007)
28. Lyell, D., Coiera, E., Chen, J., Shah, P., Magrabi, F.: How machine learning is embedded to support clinician decision making: an analysis of fda-approved medical devices. *BMJ Health Care Inform.* **28**(1) (2021)
29. Păsăreanu, C., Converse, H., Filieri, A., Gopinath, D.: On the probabilistic analysis of neural networks. In: Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 5–8 (2020)
30. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. *ACM SIGSOFT Softw. Eng. Notes* **30**(5), 263–272 (2005)
31. Singh, G., Gehr, T., Püschel, M., Vechev, M.: An abstract domain for certifying neural networks. *Proc. ACM Programm. Lang.* **3**(POPL), 1–30 (2019)
32. Su, J., Vargas, D.V., Sakurai, K.: One pixel attack for fooling deep neural networks. *IEEE Trans. Evol. Comput.* **23**(5), 828–841 (2019)
33. Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M., Kroening, D.: Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 109–119 (2018)
34. Verdoolaege, S.: The barvinok model counter (2017)
35. Wolberg, W., Mangasarian, O., Street, N., Street, W.: Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository (1995)
36. Zhang, Y., Zhao, Z., Chen, G., Song, F., Chen, T.: Bdd4bnn: a bdd-based quantitative analysis framework for binarized neural networks. In: International Conference on Computer Aided Verification, pp. 175–200 (2021)



Graph Convolutional Network Robustness Verification Algorithm Based on Dual Approximation

Dongdong An¹, Hao Zhang¹, Qin Zhao¹, Jing Liu², Jianqi Shi^{2,3(✉)},
Yanhong Huang², Yang Yang², Xu Liu³, and Shengchao Qin^{4,5}

¹ Shanghai Engineering Research Center of Intelligent Education and Bigdata,
Shanghai Normal University, Shanghai 200234, China

{andongdong, 1000548861, q.zhao}@shnu.edu.cn

² Shanghai Key Laboratory of Trustworthy Computing, East China Normal
University, Shanghai 200062, China

{jliu, yhhuang}@sei.ecnu.edu.cn, yang.yang@ntesec.ecnu.edu.cn

³ Shanghai Formal-Tech Information Technology Co., LTD., Shanghai, China
jqshi@sei.ecnu.edu.cn, xu.liu@formal-tech.com

⁴ Guangzhou Institute of Technology, Xidian University, Shaanxi, China

⁵ ICCT and ISN Laboratory, Xidian University, Shaanxi, China

Abstract. With the continuous development of Graph Neural Network (GNN) technologies, securing their robustness is crucial for their broad adoption in practical applications. Although various verification methods for training GNNs have been proposed, studies indicate that Graph Convolutional Networks (GCNs) remain vulnerable to adversarial attacks affecting both graph structure and node attributes. We propose a novel approach to verify the robustness of GCNs against perturbations in node attributes by employing a dual approximation technique to convexify nonlinear activation functions. This transformation changes the original non-convex problem into a more manageable convex forms. We start by applying linear relaxation to convert fixed-value features in each GCN layer into variables suitable for optimization. Next, we reframe the task of identifying the worst-case margin for a graph as a linear problem, which we solve using linear programming techniques. Given the discrete nature of graph data, we define a perturbation space that extends the data domain from discrete to continuous values. To improve the accuracy of the convex relaxation, we use a dual approximation algorithm to set bounds on the optimizable variables. Our method certifies the robustness of nodes against perturbations within a specified range and significantly improves verification accuracy compared to previous approaches. This method surpasses previous work in verification accuracy and is distinctively tailored to address the S-curve, an aspect less explored in prior research. Experimental results show that our method significantly refines the precision of robustness verification for GCNs.

Keywords: Robustness Verification · Linear Programming · Graph Neural Networks

1 Introduction

In recent years, the continuous development of deep learning has positioned GNNs at the forefront of research, as they combine deep learning [1] with graph data processing. With their strong representational capacity, graphs serve as a method of big data storage, and GNNs have emerged as an innovative approach for analyzing features and interactions between different entities. They have begun to offer new solutions across multiple real-world applications, such as product recommendation [2], credit assessment [3], biopharmaceuticals [4], and traffic flow prediction [3]. However, GNNs face a significant challenge: they are vulnerable to adversarial attacks. This vulnerability notably impedes their practical deployment. Numerous empirical methods aim to enhance the robustness of GNNs, inadvertently setting off an arms race in GNN attack and defence strategies. To address this, a robustness guarantee is required to ensure accurate GNN predictions within a certain range of disturbances.

Over-approximation-based GNNs have undergone extensive study [5,9]. For robustness verification, they calculate certificates for all test nodes in the testing phase to determine the number of nodes that remain secure amidst attribute perturbations. Over-approximation methods inherently introduce some overestimation, as illustrated in Sect. 2.2. An approximation that has an upper bound above the actual maximum value or a lower bound below the actual minimum introduces significant overestimation. Minimizing this overestimation is critical to preventing failures. Unlike traditional verifiable approaches, when considering GNNs, we must account for the relational dependency among graph nodes, as modifications to a single node can impact the representation of its neighbors. Furthermore, the discrete nature of graph data, specifically the challenging L_0 constraint in perturbations, and the influence of semi-supervised learning on verification, present obstacles.

To overcome the challenges above, this paper propose a method for the verifiable robustness of GCNs against node attribute perturbations. Employing dual approximation, we convert non-linear activation functions into their convex forms, allowing for efficient problem-solving. We begin with linear relaxation to turn each GCN layer’s fixed-value features into optimizable variables, then transform the problem of graph margin into a linear format solvable via linear programming. The discrete nature of graph data necessitates introducing a perturbation space, prompting a shift from discrete to continuous data domains. Further, a dual approximation algorithm calculates the constraints for these optimizable variables, thereby refining the convex relaxation. Our results emphasize the algorithm’s efficacy, demonstrating a 17.2% increase in accuracy compared to traditional training methods.

In summary, this paper’s main contributions are threefold:

1. We propose a method for verifying the robustness of GCNs relevant to node attribute perturbations, converting S-shaped activation functions into convex format and utilizing linear programming for verification.

2. We propose a linear relaxation method based on approximation domains to reduce the overestimation of function outputs, thus enhancing verification precision.
3. We have improved the accuracy of validation nodes with a maximum discrepancy of approximately 12%. We have reduced the propagation of overestimation across layers with the maximum difference reaching up to 17%.

This paper is organized as follows: Sect. 2 reviews current literature on adversarial GCNs and formal verification in this realm. Section 3 lays the foundational knowledge by including adversarial GCNs and robustness verification methods. Methods for convex relaxation of GNNs, the application of dual approximation for activation bound calculations, and linear programming for margin maximization are delineated in Sect. 4. Section 5 details experiments conducted with relevant datasets. Finally, the paper concludes by summarizing the key findings and discussing future extensions of this work.

2 Related Work

2.1 Graph Neural Network Robustness Verification Methods

The adversarial training strategies previously introduced are heuristic and demonstrate empirical benefits. However, even if current attacks fail, it remains uncertain whether adversarial examples exist. In [5], they raised a question regarding the security of nodes in a graph under any acceptable perturbation of their neighbouring nodes' attributes. To address this question, for each node v and its corresponding label y_v , they aimed to calculate the upper bound of the maximum margin loss precisely.

While [5] focused solely on perturbations to node attributes, [7] addressed scenarios where attackers manipulate only the graph structure. [10] derived robustness certificates (akin to the upper bound of the maximum margin loss) as linear functions of personalized PageRank, facilitating easier optimization. [14] also sought to prove the robustness of Graph Neural Networks under graph structure perturbations, successfully solving the certification problem using a joint-constrained bilinear programming method. Drawing inspiration from the concept of random smoothing [9, 13] achieved provable robustness for graphs under structural perturbations. Meanwhile, [8] considered the sparsity of graph data during the certification process, enhancing the efficiency and accuracy of certifications against attacks on graph features and structure. [12] introduced an immunization method to improve the provable robustness of graphs. Additionally, there has been research on the certifiable robustness of GNNs in other applications, such as community detection [11].

Our algorithm expanded the scope of verifiable models for robustness validation compared to previous work, especially [5]. Previous work could not be applied to models using sigmoid as the activation function, but our work has expanded this application and can complete robustness validation of models using sigmoid function. At the same time, our method has significantly improved

validation accuracy in multi-layer GCN models by reducing overestimation of the activation function compared to other methods targeting sigmoid.

2.2 Approximate Tightness

Neural network verification methods predicated on over-approximation inherently entail a certain degree of overestimation, as delineated in the section concerning Sect. 2.2. For an approximation of an activation function, should the maximum of its upper bound exceed the actual maximum value or the minimum of its lower bound fall below the actual minimum value, then such an approximation is deemed imprecise, introducing undue overestimation.

Reducing the overestimation of approximations is crucial for minimizing failures. The tightness concept encapsulates the approximation accuracy characteristic [23]. Significant efforts have been made to define the closest possible tightness. Definitions of tightness can be categorized into two types: neuronal and network.

1) Neuronal Tightness: The tightness of an approximation for an activation function can be measured independently. Given the activation function $\sigma(x)$ and two upper bounds $h_U(x)$ and $h'_U(x)$ over the interval $[l, u]$, $h_U(x)$ is definitively tighter than $h'_U(x)$ if $h_U(x) < h'_U(x)$ for any x in $[l, u]$ [17]. However, when $h_U(x)$ and $h'_U(x)$ intersect, their tightness becomes incomparable. An additional measure of neuronal tightness is the area of the gap between the bounds and the activation function, which is $\int_l^u (h_U(x) - \sigma(x)) dx$.

A smaller area indicates a tighter approximation [15, 18]. Notably, according to the definition in [17], an over-approximation that is tighter by one definition is also tighter by the definition in [15], and vice versa. More importantly, another metric is the output range of the linear bounds. If the approximation maintains the same output range as the activation function, it is considered the tightest [23].

2) Network Tightness: Recent research has shown that the tightness of neurons does not always guarantee that the composition of all approximations of activation functions in a network is also tight [23]. This finding explains why methods deemed the tightest based on measures of neuronal tightness achieve optimal verification results only for certain networks.

Unfortunately, finding the tightest approximation across a network has been proven to be a non-convex optimization problem, thereby incurring a high computational cost [17, 23]. From a practical standpoint, a neuron-wise tight approximation is useful if the composition of all neurons is also tight across the network.

3 Preliminaries

3.1 Neural Network Robustness Verification

Despite the challenges in verifying the correctness of DNNs, formal verification remains vital in confirming their safety-critical attributes. Among the most cru-

cial attributes is robustness, which signifies that the predictions of a neural network remain unchanged even when inputs are manipulated within a reasonable range:

Definition 1. (Neural network robustness verification). *A neural network $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is said to be robust if it accepts an input x_0 and a range Ω regarding x_0 such that for all $x \in \Omega$, $\Phi(x) = \Phi(x_0)$.*

Typically, the input region Ω centered around an input x_0 is defined by a ball with radius ϵ in the ℓ_p norm, centered at x_0 , that is: $\mathbb{B}_p(x_0, \epsilon) = \{x \mid \|x - x_0\|_p \leq \epsilon\}$.

Assuming the output label of x_0 is c , i.e., $\Phi(x_0) = c$, demonstrating the robustness of F as defined in the context of neural network robustness verification amounts to proving that for all x in Ω , and every ℓ in $L \setminus \{c\}$, $F_c(x) - F_\ell(x) > 0$. This condition verifies that for every input within the range Ω , the network's output for the correct class c surpasses that of any other class ℓ , thereby ensuring the stability of the network's predictions within this specified input domain. Hence, the verification challenge is equivalent to addressing the following optimization problem:

$$\min_{x \in \Omega} (F_c(x) - \max_{\ell \in L/c} (F_\ell(x))) \quad (1)$$

We can conclude that robustness is maintained within Ω if the result is positive. Otherwise, it implies that within Ω , there exists some variation of inputs for which there exists some $\ell' \in L \setminus \{c\}$ such that $F_{\ell'}(x') \geq F_c(x')$. This means the probability of classifying the variation as ℓ' is greater than or equal to the probability of classifying it as c . Therefore, the variation could be classified as ℓ' . This indicates that the variation does not exhibit robustness within Ω .

In this paper, given that the node features of graphs are typically measured using the l_0 norm, this necessitates a much larger ϵ radius for our sphere than is customary, where ϵ is often a fraction. Works on classical neural networks constrain adversarial examples to a small ϵ -ball around the original sample, such as under the l_∞ norm or l_2 norm [6, 24, 25], typically with $\epsilon < 1$. This is impractical in our binary setup, as $\epsilon < 1$ implies that no attribute can change. To allow for reasonable perturbations in binary/discrete settings, variations much larger than the ϵ -balls considered so far must be permitted.

The optimization problem in Eq. (1) is computationally expensive, and calculating an exact solution is often impractical. The fundamental reason for the high computational complexity lies in the non-linearity of the activation function $\sigma(x)$. Even when $\sigma(x)$ is piecewise linear, such as the commonly used ReLU ($\sigma(x) = \max(x, 0)$), the problem is NP-complete [16]. A practical solution to simplify the verification problem involves over-approximating through linear constraints and symbolic propagation, transforming it into a solvable linear programming problem that can be efficiently addressed [19, 20].

Definition 2. (Linear over-approximation). Let $\sigma(x)$ be a nonlinear function over the interval $[l, u]$, and let $h_L(x) = \alpha_L x + \beta_L$ and $h_U(x) = \alpha_U x + \beta_U$ be two linear functions with some $\alpha_L, \beta_L, \alpha_U, \beta_U \in \mathbb{R}$. $h_L(x)$ and $h_U(x)$ are referred to as the lower and upper linear bounds of $\sigma(x)$ over $[l, u]$, respectively, if $h_L(x) \leq \sigma(x) \leq h_U(x)$ applies for all x in $[l, u]$.

Based on Definition 2, we can simplify Eq. (1) into the following effective and solvable linear optimization problem. It’s important to note here that $z^{(i)}(x)$ represents an interval, not a singular value:

$$\begin{aligned} & \min \left(\min \left(z_c^{(m)}(x) \right) - \max \left(z_\ell^{(m)}(x) \right) \right) \\ \text{s.t. } & z^{(i)}(x) = W^{(i)} \hat{z}^{(i-1)}(x) + b^{(i)}, i = 1, \dots, m \\ & h_L^{(i)}(x) \leq \hat{z}^{(i)}(x) \leq h_U^{(i)}(x), i = 1, \dots, m - 1 \\ & x \in \Omega, \ell \in L/c, \hat{z}^{(0)}(x) = [x, x] \end{aligned} \quad (2)$$

In this paper, however, due to the discrete nature of the data domain, directly addressing the problem (2) remains challenging.

3.2 Slicing Matrices and Slicing Graph Convolutional Neural Networks

GNN learning can further be utilized for the embedding of sections (or graphs) in downstream applications, yet our focus is on GNNs used for node classification in semi-supervised scenarios. A representative example is the GCN. GCN for node classification take a partially labeled attribute graph as input and return the probabilities of nodes belonging to specific categories. Specifically, we assume each node belongs to one class $c \in \kappa = \{1, 2, \dots, K\}$, and the nodes in the graph are divided into labeled and unlabeled sets.

We represent an attributed graph as $\mathcal{G} = (X, A)$, where $X \in \mathbb{R}^{N \times D}$ represents the feature matrix for N nodes, with each row i being the D -dimensional feature vector associated with node i . $A \in \{0, 1\}^{N \times N}$ is the adjacency matrix of the graph.

A GCN is a multi-layer neural network represented by f_w , parameterized by W , which takes an attributed graph \mathcal{G} as input and generates a vector $Y_i = (y_i^1, \dots, y_i^K)$ for each node i , where y_i^c is the probability that node i belongs to class c .

For a GCN with L layers, the output $H_t^{(L)}$ of node t depends only on the nodes within its $L - 1$ hop neighborhood $\mathcal{N}_{L-1}(t)$. Therefore, at each step, we can “slice” the matrices X and $\hat{A}^{(l)}$ to only include the entries required for computing the output of the target node t .

$$\hat{A}^{(l)} = \hat{A}_{\mathcal{N}_{L-l}(t), \mathcal{N}_{L-l+1}(t)}^{(l)} \text{ for } l = 1, \dots, L - 1, \quad \hat{X} = X_{\mathcal{N}_{L-1}(t)} \quad (3)$$

where the set indices correspond to slicing rows and columns of the matrix. Here, $\hat{A} = A + I_{n \times n}$ is the adjacency matrix with self-loops added, \tilde{D} is the degree matrix of \hat{A} , and $\tilde{A} = \tilde{D}^{-\frac{1}{2}} \hat{A} \tilde{D}^{-\frac{1}{2}}$ is the normalized adjacency matrix. As l (i.e.,

the network depth) increases, the slicing of $A^{(l)}$ becomes smaller, and in the final step, we only need the one-hop neighbours of the target node.

With the sliced adjacency matrix and sliced feature matrix in hand, the sliced Graph Convolutional Network (GCN) can be further defined as follows:

$$\hat{H}^{(l)} = \dot{A}^{(l-1)} H^{(l-1)} W^{(l-1)} + b^{l-1} \text{ for } l = 2, \dots, L \quad (4)$$

$$H_{n_j}^{(l)} = \sigma(\hat{H}_{n_j}^{(l)}) \text{ for } l = 2, \dots, L - 1 \quad (5)$$

where $H^{(1)} = \dot{X}$, and $\hat{H}^{(l)}$ is the input before activation by the Sigmoid activation function.

3.3 Approximation Domain

Cumulative propagation primarily occurs for two reasons [26]. One apparent reason is the over-approximation of activation functions, which is inevitable but can be minimized by defining strict functions. Another reason is that over-approximation must be defined on the overestimated domain of the activation functions to ensure their robustness. As the overestimation of the domain increases, this further introduces an overestimation of the approximation values. Due to the layer dependencies in neural networks, this dual overestimation accumulates and propagates to the output layer.

We introduce the concept of an approximation domain to represent the domain of the activation function on which the over-approximation is defined.

Definition 3. (Approximation domain). *Given a neural network F and an input region $\mathbb{B}_\infty(x_0, \epsilon)$, the approximation domain for the i -th hidden neuron in the i -th layer is $[l_r^{(i)}, u_r^{(i)}]$, where,*

$$\begin{aligned} \text{s.t.} \quad & z^{(j)}(x) = W^{(j)} \hat{z}^{j-1}(x) + b^{(j)}, j \in 1, \dots, i \\ & h_L(z^{(j)}(x)) \leq \hat{z}^{(j)}(x) \leq h_U(z^{(j)}(x)), j \in 1, \dots, i - 1 \\ & x \in \mathbb{B}_\infty(x_0, \epsilon), \hat{z}^{(0)}(x) = x \end{aligned}$$

Definition 3 elaborates on the method used by existing over-approximation approaches [15, 21–23] to compute the overestimated domain of activation functions for defining their over-approximations. Given two different approximation domains $[l_r, u_r]$ and $[l'_r, u'_r]$, we say $[l_r, u_r]$ is more precise than $[l'_r, u'_r]$ if $l_r \leq l'_r$ and $u_r \geq u'_r$.

We demonstrate the interdependence between the issues of defining a strict over-approximation for an activation function and computing a precise approximation domain. This interdependence implies that a stricter over-approximation of the activation function leads to a more precise approximation domain and vice versa.

A tighter approximation obtained through the definition in [17], based on stricter bounds, results in an approximation that is also tighter by the definition based on minimum area in [15].

Based on the work in [26], we can derive tighter approximations lead to more precise approximation domains for neurons' activation functions in subsequent DNN layers and more precise approximation domains lead to tighter over-approximations of the activation function. Clearly, if we can reduce the overestimation of the approximation domain, there will be a tighter over-approximation.

4 Methods

4.1 GCN Robustness Verification Algorithm

Given that graph data often have binary or discrete features, it's not feasible to directly use norm constraints to define the perturbation space. Inspired by existing work in the domain of graph adversarial attacks [4, 5], such a perturbation space is defined by limiting the number of features that can be perturbed. Specifically, this is measured by a parameter $Q \in \mathbb{R}$ representing the change in the L_0 norm of \tilde{X} before and after perturbation. Here, \tilde{X} includes all nodes of the target node and its $L - 1$ hop neighbors because, in the field of graph learning, the embedding representation of a single node in a GCN with L layers is not only related to itself but also influenced by its $L - 1$ hop neighbors. Therefore, the parameter Q measures a global perturbation space. Additionally, to avoid excessive perturbation of any single node's features, it's necessary to define a local perturbation space further to limit the perturbation of individual node features from being too large. This constraint can be implemented through a parameter $q \in \mathbb{R}$.

Based on the above discussion, the feature perturbation space can be defined in the following form:

$$\mathcal{X}_{q,Q}(\dot{X}) = \left\{ \tilde{X} \mid \tilde{X}_{n,j} \in \{0, 1\} \wedge \|\tilde{X} - \dot{X}\|_0 \leq Q \right. \\ \left. \wedge \left\| \tilde{X}_n - \dot{X}_n \right\|_0 \leq q, \forall n \in \mathcal{N}_{L-1} \right\}. \quad (6)$$

Given a graph G , a target node t , and a GCN parameterized by θ . Let y^* denote the class of node t (for instance, provided by ground truth or predicted). Under a set of permissible perturbations to node attributes, $\hat{\chi}_{q,Q}$, the worst-case margin achievable between class y^* and any other class y is given by the following formula:

$$m^t(y^*, y) := \underset{\tilde{X}}{\text{minimize}} f_{\theta}^t(\tilde{X}, \dot{A})_{y^*} - f_{\theta}^t(\tilde{X}, \dot{A})_y \\ \text{subject to } \tilde{X} \in \hat{\chi}_{q,Q}(\dot{X}) \quad (7)$$

Building on the work presented in [5], we further obtain:

$$\begin{aligned} \hat{m}^t(y^*, y) &:= \underset{\tilde{X}, H^{(\cdot)}, \hat{H}^{(\cdot)}}{\text{minimize}} \hat{H}_{y^*}^{(L)} - \hat{H}_y^{(L)} = c^\top \hat{H}^{(L)} \\ &\text{subject to } \tilde{X} \in \hat{\chi}_{q,Q}(\tilde{X}), [H^{(\cdot)}, \hat{H}^{(\cdot)}] \in \mathcal{Z}_{q,Q}(\tilde{X}) \end{aligned} \tag{8}$$

4.2 Convex Relaxation of Activation Functions

To render the Eq. (8) convex and thus solvable efficiently, this paper necessitates convex relaxation of the *Sigmoid* activation function, adopting the approach proposed in [6]. The core idea is to treat matrices $H(\cdot)$ and $\hat{H}(\cdot)$ as optimizable variables rather than fixed values. From this perspective, Eq. (5) transforms into a set of constraints that these variables must satisfy. Subsequently, the non-linear *Sigmoid* activation function is relaxed into a convex hull.

Specifically, considering the Eq. (5), $\hat{H}_{nj}^{(l)}$ represents the input to the Sigmoid activation function. Assuming the possible perturbation space, we can obtain the upper bound $S_{nj}^{(l)}$ and lower bound $R_{nj}^{(l)}$ for this input $\hat{H}_{nj}^{(l)}$, as well as the upper bound $h_{L_{nj}}(\hat{H}_{nj}^{(l)})$ and lower bound $h_{U_{nj}}(\hat{H}_{nj}^{(l)})$ for the output $H_{nj}^{(l)}$, where:

$$\begin{aligned} h_{L_{nj}}(\hat{H}_{nj}^{(l)}) &= \alpha_{L_{nj}}(\hat{H}_{nj}^{(l)}) + \beta_{L_{nj}} \\ h_{U_{nj}}(\hat{H}_{nj}^{(l)}) &= \alpha_{U_{nj}}(\hat{H}_{nj}^{(l)}) + \beta_{U_{nj}} \end{aligned}$$

The result of the Sigmoid function can be relaxed into a convex hull as follows:

$$\begin{aligned} S_{nj}^{(l)} \leq H_{nj}^{(l)} \leq R_{nj}^{(l)} \\ h_{L_{nj}}(\hat{H}_{nj}^{(l)}) \leq \hat{H}_{nj}^{(l)} \leq h_{U_{nj}}(\hat{H}_{nj}^{(l)}) \end{aligned}$$

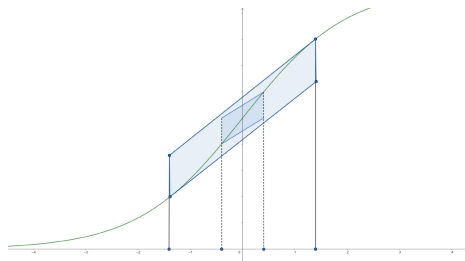


Fig. 1. Convex relaxation

This concept can be illustrated by Fig. 1. It can be observed that H is no longer a deterministic output of the Sigmoid activation function but rather an optimizable variable constrained within a certain range.

4.3 Activation Function's Upper and Lower Bounds

A key component of our method involves defining the computation of relaxed boundaries $R^{(l)}$ and $S^{(l)}$ for activations within a GCN, which remains to be specified. Similarly, existing bounds for classical neural networks are not applicable as they neither consider L_0 constraints nor account for adjacent instances. Obtaining good upper and lower bounds is crucial for achieving robustness proofs, as tighter upper bounds lead to smaller relaxation errors in GNN activations.

Although through Eq. (8), we relaxed the condition of the discreteness of node attributes X in the linear programming framework, it has been demonstrated that the binary nature of data can be leveraged for the boundaries. More precisely, for each node $m \in N_{L-2}(t)$, we compute the upper bound $S_{mj}^{(2)}$ for potential dimension j on the second layer:

$$S_{mj}^{(2)} = \text{sum_top_Q} ([A_{mn}^{(1)} \hat{S}_{nji}^{(2)}]_{n \in N_1(m), i \in 1, \dots, q}) + \dot{H}_{mj}^{(2)} \quad (9)$$

$$\hat{S}_{nji}^{(2)} = \text{i-the_largest} \left((1 - \dot{X}_n) \odot [W_{:j}^{(1)}]_+ + \dot{X}_n \odot [W_{:j}^{(1)}]_- \right) \quad (10)$$

In the context provided, i -the_largest(\cdot) refers to selecting the i^{th} largest element from the corresponding vector, and $\text{sum_top_Q}(\cdot)$ denotes the sum of the top Q largest elements from the corresponding list. The first term in the sum of Eq. (9) represents the upper bound of the change/increase in the activation of node and hidden dimension j in the first hidden layer for any permissible perturbation on the attributes \dot{X} . The second term is the hidden activation obtained from the (calm) input X .

Similarly, for the lower bound, we use the following approach:

$$R_{mj}^{(2)} = - \text{sum_top_Q} ([A_{mn}^{(1)} \hat{R}_{nji}^{(2)}]_{n \in N_1(m), i \in 1, \dots, q}) + \dot{H}_{mj}^{(2)} \quad (11)$$

$$\hat{R}_{nji}^{(2)} = \text{i-the_largest} \left(\dot{X}_n \odot [W_{:j}^{(1)}]_+ + (1 - \dot{X}_n) \odot [W_{:j}^{(1)}]_- \right) \quad (12)$$

Since their inputs are no longer binary for the subsequent layers, we adopt the bounds proposed in [25] for their calculation. Extending these to GCN, we obtain:

$$\begin{aligned} R^{(l)} &= \dot{A}^{(l-1)} \left(R^{(l-1)} [W_+^{(l-1)}] - S^{(l-1)} [W^{(l-1)}]_- \right) \\ S^{(l)} &= \dot{A}^{(l-1)} \left(S^{(l-1)} [W_+^{(l-1)}] - R^{(l-1)} [W^{(l-1)}]_- \right) \end{aligned} \quad (13)$$

for $l = 3, \dots, L - 1$

4.4 Utilizing Under-Approximation for Tight over-Approximation in GCN Robustness Verification

Given that the tightness of convex relaxation approximations significantly influences the precision with which we can judge node robustness, it is crucial to enhance tightness as much as possible. We utilize the dual approximation approach introduced in [26] to define tight over-approximations of activation functions guided by under-approximation.

For each activation function, we calculate both an overestimated and an underestimated approximation domain, represented as $[l_{over}, u_{over}]$ and $[l_{under}, u_{under}]$, respectively. The underestimated domain provides valuable information for defining tight over-approximations. Let $h(x)$ be a linear lower or upper bound for σ over the interval $[l_{over}, u_{over}]$. Suppose it satisfies the conditions in Definition 2. In that case, we consider it as a lower or upper bound for the linear over-approximation on the approximation domain $[l_{over}, u_{over}]$. According to Theorem ??, we can define a tighter bound on $[l_{over}, u_{over}]$ than one defined on $[l_{over}, u_{over}]$, and ensure an effective over-approximation boundary on $[l_{over}, u_{over}]$. Therefore, we refer to this as the dual approximation method using the underestimated domain to guarantee the tightness of over-approximations and the approximation domain to ensure the robustness of over-approximations.

Assuming the input’s lower approximation domain is $[l_{under}, u_{under}]$ and the upper approximation interval is $[l_{over}, u_{over}]$, as described in [21], we consider three scenarios based on the relationship between $\sigma'(l_{over})$, $\sigma'(u_{over})$, and k , where $k = \frac{\sigma(u_{over}) - \sigma(l_{over})}{u_{over} - l_{over}}$. Algorithm ?? displays the pseudo-code for the dual approximation.

1. When $\sigma'(l_{over}) < k < \sigma'(u_{over})$, the line connecting the two endpoints serves as the upper bound. For the lower bound, if it is feasible, the tangent at the point is taken; otherwise, the tangent intersecting with $(u_{over}, \sigma(u_{over}))$ is chosen.
2. When $\sigma'(l_{over}) > k > \sigma'(u_{over})$, the line connecting the two endpoints serves as the lower bound. For the upper bound, if feasible, the tangent line at the point is taken; otherwise, the tangent intersecting with $(l_{under}, \sigma(l_{under}))$ is chosen.
3. When $\sigma'(l_{over}) < k$ and $\sigma'(u_{over}) < k$, we first consider the upper bound. If the tangent of the polynomial is reliable, we choose it as the upper bound; otherwise, we select the tangent that intersects with $(l_{under}, \sigma(l_{under}))$. Then, we consider the lower bound. If it is reasonable, the tangent at that point is taken; otherwise, the tangent intersecting with $(u_{over}, \sigma(u_{over}))$ is chosen.

5 Experiment

5.1 Experimental Dataset

This paper experiments¹ on two graph datasets widely used in graph learning research: Cora and Karate. For each node in these datasets, 1% of the features are

¹ <https://github.com/cloud-rise-world/VRP.git>

allowed to be perturbed, that is, $q = 0.01d$, where d represents the dimensionality of the input features. Different values of Q are set for global perturbations to observe their effects.

5.2 Experimental Setup

To ensure a fair comparison between the dual approximation algorithm, which requires the use of the Monte Carlo method to generate multiple samples and thus consumes more resources, and the NeWise method, the number of generated samples by the Monte Carlo method was reduced in the experiments. The size of S_n , the sample set, was set to 10% of the dataset size.

For the same GCN model, we apply both the dual approximation robustness verification algorithm and the NeWise approximation robustness verification algorithm to validate its robustness. The comparison between these two algorithms is conducted by examining the proportion of robust nodes relative to the total number of nodes, assessing the effectiveness of each algorithm.

Additionally, we consider different values of L to compare the robust nodes obtained by the dual approximation algorithm and the NeWise algorithm. This comparison helps to highlight the advantages of the dual approximation algorithm in reducing overestimation.

5.3 Verify the Robustness of GCN

Robustness verification of the GCN model is conducted using the dual approximation robustness verification algorithm and the NeWise approximation robustness verification algorithm proposed in this paper.

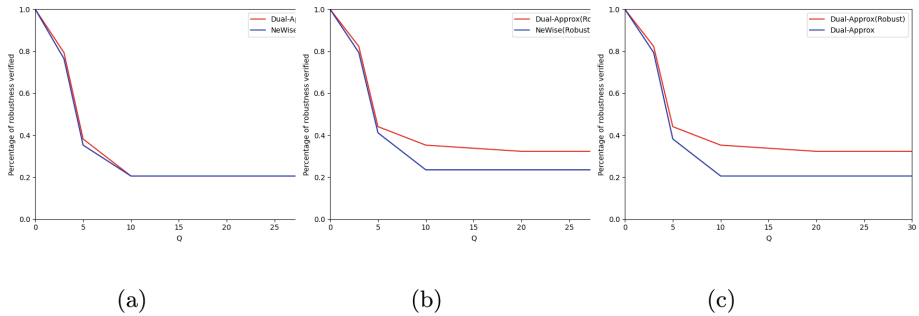


Fig. 2. $L = 2$ changes in robust nodes

Figure 2 illustrates how the GCN model performs on the Karate dataset with $L = 2$. The blue and orange lines show the proportion of robust nodes identified by the dual approximation robustness verification algorithm and the NeWise approximation algorithm, respectively.

Figure 2(a) compares the number of robust nodes identified by both algorithms before the GCN model undergoes robustness training. The vertical gap between the two curves shows that the dual approximation algorithm finds more robust nodes than the NeWise algorithm, with the largest difference being around 12%. This highlights that the relaxation techniques and bounds on non-linear activation functions proposed in this paper offer more precise results.

After robustness training is applied to the GCN, the model is re-evaluated using both algorithms, as shown in Fig. 2(b). The results show a significant increase in the number of robust nodes, with the dual approximation algorithm consistently identifying more than the NeWise algorithm, maintaining a 12% difference at its peak. This confirms the effectiveness of the proposed verification method for GCN robustness.

Figure 2(c) contrasts the number of robust nodes identified by the dual approximation algorithm before and after the model’s robustness training. It’s clear that after training, more robust nodes are identified, reinforcing the effectiveness of the robustness training and the proposed validation method for GCN robustness.

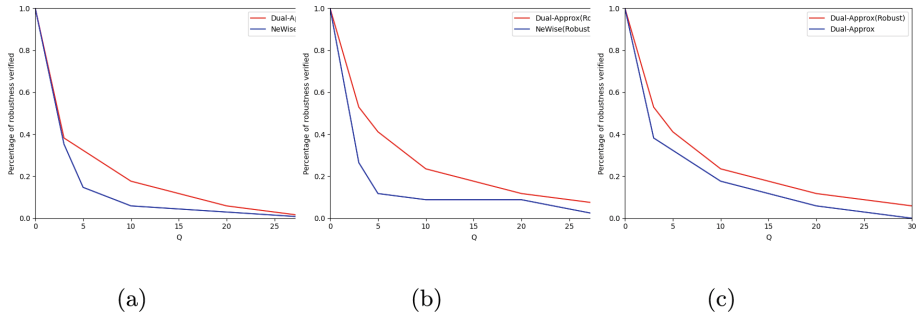


Fig. 3. $L = 3$ changes in robust nodes

Figure 3 shows how the GCN performs with $L = 3$. In this case, fewer robust nodes are identified compared to the $L = 2$ scenario. The vertical gap between the two curves is larger than with $L = 2$, reaching a maximum of 17%. This suggests that increasing the number of layers makes the model more fragile, but the dual approximation algorithm reduces this overestimation.

Table 1 shows that as the perturbation space Q increases, the proportion of robust nodes decreases, while the proportion of non-robust nodes rises. For example, with $L = 2$ and $Q = 3$, 79.41% of nodes are robust without robustness training, increasing to 82.35% after training. At $Q = 10$, only 20.59% of nodes are robust without training, but this improves to 35.29% after training. This further demonstrates the effectiveness of robustness training and shows that GCNs are sensitive to small input changes.

Relative to the $L = 2$ model, the $L = 3$ model appears more fragile under perturbation. For instance, when $Q = 20$, 20.59% of nodes remain robust in the

Table 1. Experimental result

L	Model	Q = 3		Q = 5		Q = 10		Q = 20	
		Dual	NeWise	Dual	NeWise	Dual	NeWise	Dual	NeWise
L = 2	<i>Normal</i>	79.41%	76.47%	38.24%	35.29%	20.59%	20.59%	20.59%	20.59%
	<i>Robust</i>	82.35%	79.41%	44.12%	41.18%	35.29%	23.53%	32.35%	23.53%
L = 3	<i>Normal</i>	38.24%	35.29%	32.35%	14.71%	17.65%	5.88%	5.88%	2.94%
	<i>Robust</i>	52.94%	26.47%	41.18%	11.76%	23.53%	8.82%	11.76%	8.82%

$L = 2$ model, but only 5.88% in the $L = 3$ model. This highlights how overestimations propagate through the model layers and worsen with an increase in depth, thereby affecting the verification of robust nodes. The dual approximation algorithm performs better than the NeWise algorithm under the $L = 3$ model conditions. For example, when $Q = 10$ and as layer depth increases, the proportion of robust nodes identified by the dual approximation algorithm drops from 20.59% to 17.65%. In contrast, for the NeWise algorithm, it falls from 17.65% to 5.88%. The dual approximation algorithm is significantly less affected by the increase in layer depth than the NeWise algorithm, reflecting its superior ability to mitigate overestimations.

6 Conclusion

This paper introduced a GCN robustness verification algorithm to counter node feature adversarial attacks. The issue of robustness within graph neural networks is initially defined and subsequently formulated as a linear programming problem. A dual approximation method is utilized to address the prevalent issue of overestimation that arises during the convex relaxation of neural networks. The algorithm verifies the robustness of GCN models independently of particular attack algorithms, data labeling, or downstream tasks. Experimentation validates the method’s efficacy, with comparative studies further affirming the algorithm’s capacity to minimize overestimation, thereby highlighting the significance of the proposed GCN robustness verification algorithm. Currently, the algorithm does not account for robustness under structural perturbations. Future work will extend to include structural perturbations, aiming to contribute to the development of a GCN algorithm that ensures robustness against a diverse array of attacks.

Acknowledgment. This work was sponsored in part by the National Key Research and Development Program of China under Grant No. 2022YFB4501704 and 2022YFC3302600; in part by the National Natural Science Foundation Youth Fund under Grant No. 62302308; in part by the National Natural Science Foundation of China under Grant No. 61972150, 62132014, 62302308, U2142206, 62372300, and 61702333; in part by the Shanghai Engineering Research Center of Intelligent Education and Big Data; and in part by the Research Base of Online Education for Shanghai Middle and Primary Schools.

References

1. Joardar, B.K., Arka, A.I., Doppa, J.R., Pande, P.P., Li, H., Chakrabarty, K.: Heterogeneous manycore architectures enabled by processing-in-memory for deep learning: from CNNs to GNNs: (ICCAD Special Session Paper). In: 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), November 2021, pp. 1–7. <https://doi.org/10.1109/ICCAD51958.2021.9643559>
2. Niu, X., et al.: A dual heterogeneous graph attention network to improve long-tail performance for shop search in E-Commerce. In: Gupta, R., Liu, Y., Tang, J., Prakash, B.A. (eds.) KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23–27, 2020. ACM, pp. 3405–3415. <https://doi.org/10.1145/3394486.3403393>
3. Bui, K.-H.N., Cho, J., Yi, H.: Spatial-temporal graph neural network for traffic forecasting: an overview and open research issues. *Appl. Intell.* **52**(3), 2763–2774 (2022)
4. Zügner, D., Akbarnejad, A., Günnemann, S.: Adversarial attacks on neural networks for graph data. In: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. ACM, London, pp. 2847–2856 (2018)
5. Zügner, D., Günnemann, S.: Certifiable robustness and robust training for graph convolutional networks. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. Anchorage, USA, pp. 246–256. ACM (2019)
6. Wong, E., Kolter, Z.: Provable defenses against adversarial examples via the convex outer adversarial polytope. In: Proceedings of the 35th International Conference on Machine Learning. Stockholmsmässan, Stockholm, Sweden: PMLR, pp. 5283–5292 (2018)
7. Bojchevski, A., Günnemann, S.: Certifiable robustness to graph perturbations. In: Advances in Neural Information Processing Systems, pp. 8317–8328 (2019)
8. Bojchevski, A., Klicpera, J., Günnemann, S.: Efficient robustness certificates for discrete data: Sparsityaware randomized smoothing for graphs, images and more. arXiv preprint [arXiv:2008.12952](https://arxiv.org/abs/2008.12952) (2020)
9. Cohen, J.M., Rosenfeld, E., Kolter, J.Z.: Certified adversarial robustness via randomized smoothing. arXiv preprint [arXiv:1902.02918](https://arxiv.org/abs/1902.02918) (2019)
10. Jeh, G., Widom, J.: Scaling personalized web search. In: Proceedings of the 12th International Conference on World Wide Web, pp. 271–279 (2003)
11. Jia, J., Wang, B., Cao, X., Gong, N.Z.: Certified robustness of community detection against adversarial structural perturbation via randomized smoothing. arXiv preprint [arXiv:2002.03421](https://arxiv.org/abs/2002.03421) (2020)
12. Tao, S., Shen, H., Cao, Q., Hou, L., Cheng, X.: Adversarial immunization for improving certifiable robustness on graphs. arXiv preprint [arXiv:2007.09647](https://arxiv.org/abs/2007.09647) (2020)
13. Wang, B., Jia, J., Cao, X., Gong, N.Z.: Certified robustness of graph neural networks against adversarial structural perturbation. arXiv preprint [arXiv:2008.10715](https://arxiv.org/abs/2008.10715) (2020)
14. Zügner, D., Günnemann, S.: Certifiable robustness of graph convolutional networks under structure perturbations. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 1656–1665 (2020)
15. Henriksen, P., Lomuscio, A.: Efficient neural network verification via adaptive refinement and adversarial search. In: European Association for Artificial Intelligence (ECAI'20), pp. 2513–2520 (2020)

16. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: an efficient SMT solver for verifying deep neural networks. In: Majumdar, R., Kunčák, V. (eds.) CAV 2017. LNCS, vol. 10426, pp. 97–117. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_5
17. Lyu, Z., Ko, C.-Y., Kong, Z., Wong, N., Lin, D., Daniel, L.: Fastened CROWN: tightened neural network robustness certificates. In: AAAI Conference on Artificial Intelligence (AAAI'20), pp. 5037–5044 (2020)
18. Müller, M., Makarchuk, G., Singh, G., Püschel, M., Vechev, M.T.: PRIMA: general and precise neural network certification via scalable convex hull approximations. *Proc. ACM Program. Lang.* **6**(2022), 1–33 (2022)
19. Wang, S., Pei, K., Whitehouse, J., Yang, J., Jana, S.: Formal security analysis of neural networks using symbolic intervals. In: USENIX Security'18. 1599–1614 (2018)
20. Wang, Z., Albarghouthi, A., Prakriya, G., Jha, S.: Interval universal approximation for neural networks. *Proc. ACM Program. Lang.* **6**, POPL, pp. 1–29 (2022)
21. Wu, Y., Zhang, M.: Tightening robustness verification of convolutional neural networks with fine-grained linear approximation. In: AAAI Conference on Artificial Intelligence (AAAI'21), pp. 11674–11681
22. Zhang, H., Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., Daniel, L.: Efficient neural network robustness certification with general activation functions. In: Advances in Neural Information Processing Systems (NeurIPS'18), pp. 4944–4953 (2018)
23. Zhang, Z., Wu, Y., Liu, S., Liu, J., Zhang, M.: Provably tightest linear approximation for robustness verification of sigmoid-like neural networks. In: IEEE/ACM International Conference on Automated Software Engineering (ASE'22), pp. 80:1–80:13. ACM (2022)
24. Croce, F., Andriushchenko, M., Hein, M.: Provable robustness of relu networks via maximization of linear regions. In: AISTATS (2018)
25. Raghunathan, A., Steinhardt, J., Liang, P.S.: Semidefinite relaxations for certifying robustness to adversarial examples. In: NIPS (2018)
26. Xue, Z., Liu, S., Zhang, Z., Wu, Y., Zhang, M.: A tale of two approximations: tightening over-approximation for dnn robustness verification via under-approximation. In: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023), pp. 1182–1194. Association for Computing Machinery, New York (2023). <https://doi.org/10.1145/3597926.3598127>



Formal Kinematic Analysis of Epicyclic Bevel Gear Trains

Kubra Aksoy^(✉), Adnan Rashid, and Sofiène Tahar

Department of Electrical and Computer Engineering, Concordia University,
Montreal, QC, Canada

{k_aksoy,rashid,tahar}@ece.concordia.ca

Abstract. Epicyclic Bevel Gear Trains (EBGTs) play a vital role in providing highly efficient solutions for power transmissions between shafts in various engineering applications, such as wind turbines and jet airplane engines. The kinematic analysis of EBGTs involves identifying fundamental cycles, and utilizing screw theory to understand the velocities and relative motion of the system's components. In this paper, we propose to use higher-order-logic theorem proving for the formal kinematic analysis of EBGTs. In particular, we formalize a directed graph representation of EBGT systems, consisting of links and joints (pairs). Next, we formalize a corresponding cycle matrix form of the EBGT in order to analyze fundamental cycles in the graph. Moreover, we formalize the linear and angular velocities of the joint components in the systems using various notions of screw theory, such as screw and twist. We use the above to formally verify the kinematic equations providing a sound relationship between the relative angular joint velocities. In order to illustrate the utility of our proposed formalization, we formally analyze the Bendix wrist, which is a well-known industrial geared mechanism, by providing a verified solution of its kinematic equations.

Keywords: Epicyclic Gear Trains · Kinematic Analysis · Cycle Matrix · Screw Theory · Higher-Order Logic · Theorem Proving · Isabelle/HOL

1 Introduction

Epicyclic Bevel Gear Trains (EBGTs) [22], known as transmission mechanisms, are composed of gear pairs that have intersecting axes, and at least one gear axis is in circular motion with respect to the ground/fixed link of the mechanism. EBGTs are integrated to a variety of mechanisms for the purpose of transmitting rotational motion and/or adapting the speed of various components of mechanisms. EBGTs are used in a wide range of engineering and mechanical applications, such as automotive, avionics, aerospace and renewable energy, thanks to their advantages like altered speed ratios and higher efficiency of the power transmission. For instance, EBGTs are of key importance in the design of robotic wrists for power transmission, which assist in reducing the weight and inertia of

a robotic manipulator and thus enhancing the efficiency of the mechanism [18]. Due to their aforementioned abilities and usage in safety-critical applications, e.g., wind turbines and jet airplane engines, their kinematic analysis becomes quite significant since the occurrence of any unexpected behavior may result in financial loss and even fatalities.

To perform a kinematic analysis of an EBGT mechanism, we need to develop a set of kinematic equations capturing the relative angular velocities¹ of various components of the system, such as joints and links, which are further analyzed by finding out their solutions providing a relationship between the output and input velocities. There are several techniques used for the kinematic analysis including tabular methods [14], train value methods [4] and graph theory methods [20, 21]. For instance, one of the commonly used methods in textbooks is the *tabular method* based on Willis' inversion method [24]. However, this method is applicable to non-parallel axis gear trains mechanisms only. Similarly, the *train value method* solely focuses on the overall angular velocity ratios, and thus cannot deal with the velocities of intermediate links. On the other hand, the *graph theory* with the fundamental cycle concept [16] is suitable for systems with plenty of gears and multitude Degree of Freedoms (DOF). Here, the analysis starts with a synthesis of the kinematic structure of the system using fundamental cycles, from which a cycle matrix, as an algebraic representation, is used to capture the relationship between the edges and the cycles. Next, by utilizing the *screw theory* [5], the velocity of each joint is represented by screws and twists based on the fixed frame of each component. Then, the orthogonality conditions are established using the cycle matrix and screw/twist matrices in order to derive the kinematic equations. Finally, these equations for the relative angular velocities of joints are analyzed to conclude the analysis.

Conventionally, EBGTs are analyzed using paper-and-pencil proof and computer-based simulation techniques. However, the former is prone to human-error, while the latter frequently relies on unverified numerical algorithms in the core of the associated tools that can introduce approximation errors, leading to potential inaccuracies in the results. In consideration of these limitations, conventional methods do not provide the necessary level of accuracy and precision required for a comprehensive analysis of systems. On the other hand, formal methods, such as higher-order-logic (HOL) theorem proving, provides a more rigorous approach by constructing computer-based mathematical modelling of systems and verifying its properties through logical reasoning. HOL theorem proving is hence well-suited for the formal modeling and analysis of EBGT mechanisms since it provides soundness and a high level of expressiveness.

In this paper, we propose to use the Isabelle/HOL [13] for performing the kinematic analysis of EBGTs. In particular, we formalize cycle matrix based directed graph models of EBGT mechanisms using higher-order logic theorem proving. In general, an EBGT can be modeled using either an undirected graph with adjacency matrix or a directed graph with cycle matrix. Unlike undirected

¹ Relative angular velocity, or rotational velocity, is the difference between the rotational speeds of two links.

graphs with adjacency matrices, a directed graph based analysis does not require additional rules to identify graph cycles, which are obtained using a cycle matrix that is further used to generate the kinematic equations of the system [19]. Furthermore, directed graphs can model symmetric and asymmetric relationships between components, making them more generic. Therefore, in this paper, we use a directed-graph based modeling approach to analyze EGBT mechanisms. Next, we formalize the screw for describing spatial geometry of local frames assigned to joints, which provides rotational and translational motion of joints. Similarly, we formalize the twist vector to model angular and linear velocity of joints. In the next step, we utilize the above formalization alongside the orthogonality condition to formalize a set of kinematic equations capturing the relative angular velocities. To demonstrate the utility of our proposed formalization, we formally analyze a Bendix wrist mechanism [19] by verifying its kinematics using Isabelle/HOL. To the best of our knowledge, there exists no formal analysis of EGBT systems that uses graph-based cycle matrices and screw theory.

The remainder of the paper is organized as follows: Sect. 2 discusses an overview of the related work regarding the formal kinematic analysis of engineering and physical systems, topology matrices and screw theory. We present a foundational formalization of EGBTs analysis in Sect. 3. As an application, in Sect. 4, we develop the formal kinematic analysis of the Bendix wrist mechanism in Isabelle/HOL. Finally, Sect. 5 concludes the paper with pointers to some future directions.

2 Related Work

Higher-order-logic theorem proving has been used for the formal kinematic analysis of engineering and physical systems. For instance, Farooq et al. [10] used the HOL Light theorem prover to formally analyze the kinematics of two-link planar manipulators. Similarly, Rashid et al. [15] formally analyzed the dynamics of robotic cell injection systems up to 4-DOF using HOL Light. Chen et al. [7] also used HOL Light to formalize a camera pose estimation algorithm based on Rodrigues formula for robotic systems. Moreover, Wang et al. [23] formally verified the inverse kinematics of a three-fingered dexterous hand by analyzing the Paden-Kahan-sub-problem based on the screw theory in HOL Light.

On the other hand, Affeldt et al. [1] developed some geometrical foundations in 3-Dimensions (3D), including rotation matrices, screw motion as well as Denavit-Hartenverg (D-H) convention for forward kinematics of robot manipulators using the Coq theorem prover. Similarly, Wu et al. [25] formalized a Jacobian matrix to perform the forward kinematic analysis of a 3-DOF planar robot manipulator in the HOL4 theorem prover. Later, Shi et al. [17] extended this work by formally verifying the kinematic Jacobian for serial manipulators using the screw-based methods. In particular, the authors formalized twists to represent relative motion of a rigid body using exponential mapping. Recently, Xie et al. [26] used Coq to formalize coordinate transformation for robots, especially for spiral motion of rigid bodies, using Rodrigues formula and homogeneous matrices.

The aforementioned methods, including the screw-based approach, are focused on formally analyzing serial robotic manipulators, and do not consider any topological aspects of the systems, which are required for a comprehensive analysis of EGBTs.

Regarding topological matrices, there exist only a few formalizations in higher-order-logic theorem provers. For example, Heras et al. [11] formalized incidence matrices for undirected graph-based representations using Coq and used it to formally analyze 2D digital image processing systems. Recently, Edmonds et al. [9] formalized incidence matrices in Isabelle/HOL to represent a design in order to verify the Fisher's inequality. However, these contributions are able to only analyze systems represented by *undirected graphs*. Moreover, they do not focus on the kinematic analysis of EGBTs mechanisms, which is the scope of the current paper.

3 Formalization of EGBTs Analysis Foundations

3.1 Formalization of Cycle Matrices

A directed graph is defined by an ordered pair $DG = (N, E)$, where N represents a set of nodes and E is a set of edges, where each of them is a pair of distinct nodes. For the case of EGBT mechanisms, in a mechanical digraph (directed graph) representation, links are represented by nodes and connectors between the links, called joints/pairs are represented by directed edges. The mechanical digraph of the mechanisms with cycles can be algebraically represented as a matrix, called *cycle matrix*. This matrix captures the relationship between the cycles and joints of a mechanism represented by a (mechanical) directed graph. A cycle matrix is mathematically defined as follows [8]:

Definition 1. *Cycle Matrix of a Directed Graph*

Consider a set of edges $\{e_1, \dots, e_n\}$ and cycles $\{L_1, \dots, L_l\}$. \mathcal{C} is an $l \times n$ cycle matrix of the directed graph, such that

$$C_{i,j} = \begin{cases} 1 & \text{if } e_j \in L_i, \text{ and the direction of } e_j \text{ and } L_i \text{ are the same} \\ -1 & \text{if } e_j \in L_i, \text{ and the direction of } e_j \text{ and } L_i \text{ are opposite} \\ 0 & \text{if } e_j \notin L_i \end{cases}$$

Here a cycle of a graph is a closed path, defined as a finite sequence of distinct edges where the start and the end nodes of the path are the same. It is worth to note that in this paper, we consider mechanical digraphs EGBTs without self loops, and having distinct cycles in a cycle-basis². This concept, known as *fundamental cycles*, enables the analysis of the system on any cycle basis, which is sufficient to understand the kinematics of the entire mechanism. Moreover, the cycle matrix of the mechanism in this concept holds an important property

² A cycle basis in a directed graph is defined as the set of independent cycles.

that the number of gear pairs is the same as the number of cycles in a cycle basis.

In order to formalize a cycle matrix in Isabelle/HOL, we first formally model a mechanical digraph utilizing locale modules [6] in Isabelle/HOL. A locale module provides a series of context elements, needed to structure abstract algebraic concepts. These elements, namely parameters and assumptions, are declared using the keywords `fixes` and `assumes`, respectively. Locales can be expanded by adding new parameters, definitions and assumption into existing ones, which makes them flexible and reusable.

We now introduce a locale `mech_digraph` consisting of a list of nodes ($\mathcal{N}s$) and a list of edges ($\mathcal{E}s$), as well as the component relationships as well-formed assumptions³. It is worth mentioning that we chose the parameter types as `real` for the purpose of labeling the nodes and thus edges.

```

locale mech_digraph =
fixes nodes_list :: nodes ( $\mathcal{N}s$ ) and edges_list :: edges ( $\mathcal{E}s$ )
assumes mechdg_wf:  $e \in \text{set } \mathcal{E}s \implies \text{fst } e \in \text{set } \mathcal{N}s \wedge \text{snd } e \in \text{set } \mathcal{N}s \wedge \text{fst } e \neq \text{snd } e$ 
assumes distincts: distinct  $\mathcal{N}s$  distinct  $\mathcal{E}s$ 

```

where the function `set` accepts a list of nodes and edges and returns a set. Similarly, the functions `fst` and `snd` accept a pair, and extract its first and second elements, respectively. The function `distinct` takes a list and ensures that elements of the list are disjoint. Furthermore, the assumption `mechdg_wf` ensures that the digraph has no self-loop. Next, we formalize the system with cycles by adding cycle parameter and well-formed assumptions on the locale `nempty_mechdg` by ensuring a valid digraph with a non-empty list of nodes and edges.

```

locale cycle_system = nempty_mechdg +
fixes cycle_basis :: edges list ( $\mathcal{L}s$ )
assumes wf_1:  $ls \in \text{set } \mathcal{L}s \implies \text{set } ls \subseteq \text{symcl } \mathcal{E} \wedge \text{length } ls \leq \text{length } \mathcal{E}s$ 
assumes wf_2:  $ls \in \text{set } \mathcal{L}s \implies \text{cycle } ls \wedge \text{cycle } (\text{reverse } ls)$ 
and distinct: distinct  $\mathcal{L}s$ 

```

where `symcl` accepts a Cartesian set and guarantees that this set contains elements with their reversed version. The assumption `wf_1` provides a validity of every cycle by ensuring that every element of the `cycle_basis` $\mathcal{L}s$ is a subset of the symmetric relation of edges. It also makes sure that the size of the cycle cannot be larger than the number of edges in a graph. Similarly, `wf_2` guarantees that every element of the cycle list is a cycle and its reverse is also a cycle. The assumption `distinct` ensures the non-repetition of cycles in the cycle basis. Additionally, we formally define a `nempty_cycle_system` locale on top of the `cycle_system`, which ensures that the cycle system has at least one cycle in the cycle basis.

³ We abbreviate `real` as `node`, `real list` as `nodes`, `real×real` as `edge`, and `(real×real) list` as `edges` for a better readability.

```

locale nempty_cycle_system = cycle_system +
    assumes cycle_basis_nempty:  $\mathcal{L}s \neq []$ 
    
```

We now describe the relationship between the directions of a cycle and its corresponding edges. The direction of edges in a cycle is the same as that of a cycle, called positively oriented cycle, whereas the cycle is said to be negatively oriented with the edge if the edge is in the reversed cycle. For instance, a positively oriented cycle is formalized as follows:

```

definition in_pos_cycle  $x\ es \equiv (x, es) \in C_p$ 
    
```

where C_p is a set of pairs consisting of the edge and the cycle, formalized as:

```

definition  $C_p \equiv \{(x, es). x \in \text{set } \mathcal{E}s \wedge es \in \text{set } \mathcal{L}s \wedge x \in \text{set } es\}$ 
    
```

Similarly, the formalization of the relation between the edge and the cycle where the edge is negatively oriented with the cycle is given as:

```

definition in_neg_cycle  $x\ es \equiv (x, es) \in C_n$ 
    
```

where C_n is a set of pairs consisting of the edge and the cycle

```

definition  $C_n \equiv \{(x, es). x \in \text{set } \mathcal{E}s \wedge es \in \text{set } \mathcal{L}s \wedge x \notin \text{set } es$ 
     $\wedge x \in \text{set } (\text{reverse } es)\}$ 
    
```

Here, the function `reverse` takes a pair list and reverses its order by swapping the elements of each pair in the list. Therefore, we obtain the concept of negatively oriented cycles using the function `reverse`. Note that we explicitly indicate the direction of the cycle while assuming each edge is positively directed. Next, the cycle matrix is formalized in Isabelle/HOL as follows:

```

definition cycle_matrix :: edges list  $\Rightarrow$  edges  $\Rightarrow$  real mat
    where cycle_matrix  $\mathcal{L}s\ \mathcal{E}s \equiv \text{mat } (\text{length } \mathcal{L}s) (\text{length } \mathcal{E}s)$ 
     $(\lambda(k, j). \text{if } (\mathcal{E}s!j) \in \text{set } (\mathcal{L}s!k) \text{ then } 1 \text{ else}$ 
     $\text{if } (\mathcal{E}s!j) \in \text{set } (\text{reverse } (\mathcal{L}s!k)) \text{ then } -1 \text{ else } 0)$ 
    
```

3.2 Formalization of Screw Theory Notions

The screw theory provides a unified framework for describing the spatial displacement (screw motion) of a rigid body, encompassing both rotational and translational components. Similarly, the linear and angular velocities of a rigid body can be expressed within a single concept using a screw, called twist. A screw [5] is mathematically defined as a dual vector consisting of two three-dimensional vectors, where the first vector represents the direction vector of a line (screw axis) and the second describes the moment vector specifying the translation along the screw axis. When a screw has zero pitch⁴, resulting in pure

⁴ A pitch is a scalar quantity that describes the ratio between the translational and rotational part of the screw.

rotation without translation, its six elements become mathematically equivalent to the Plücker coordinates [12]. In the analysis of EBGTs, a screw is analogous to Plücker coordinates that can define the spatial geometry of the axis z_k of the locale frames attached to the pair k [20]. The screw $\mathbf{u}_{c,k}^0$ is mathematically expressed as follows:

$$\mathbf{u}_{c,k}^0 = \left(\frac{\mathbf{u}_k^0}{\mathbf{I}_{c,k}^0 \times \mathbf{u}_k^0} \right) \tag{1}$$

where the subscripts k and c refer to all pairs⁵ and gear pairs, respectively. Similarly, the superscript 0 denotes that the components of the vectors in a screw are measured with respect to the base frame. The first vector in the screw, \mathbf{u}_k^0 , describes the orientation of the unit vectors of a frame pair k with respect to the base frame 0. The first vector \mathbf{u}_k^0 can mathematically be derived using a *direction cosines matrix* as follows [20]:

$$\mathbf{u}_k^0 = \Theta_{0,k} \cdot \mathbf{u} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\varphi_{z_0,z_k} & \sin\varphi_{z_0,z_k} \\ 0 & -\sin\varphi_{z_0,z_k} & \cos\varphi_{z_0,z_k} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{2}$$

where \mathbf{u} is the unit vector along the z -axis. Moreover, $\Theta_{0,k}$ is a direction cosines matrix that describes the instantaneous orientation of the z -axis in the base frame with respect to the z -axes of all pairs frames, and thus provides their rotation about the respective axes. In addition, φ_{z_0,z_k} presents the angle between the z -axes of the base and the pair frames and is equally represented by $\widehat{(z_0, z_k)}$. By simplifying Eq. (2), we obtain the first vectors of screw for EBGTs as follows:

$$\mathbf{u}_k^0 = \left(\begin{array}{c} 0 \\ \sin(\widehat{z_0, z_k}) \\ \cos(\widehat{z_0, z_k}) \end{array} \right) \tag{3}$$

Similarly, the second vector in Eq. (1), $\mathbf{I}_{c,k}^0 \times \mathbf{u}_k^0$, called the moment of the unit vector \mathbf{u}_k^0 , is defined as the cross product of two vectors such that

$$\mathbf{I}_{c,k}^0 \times \mathbf{u}_k^0 = \mathbf{r}_{c,k}^0 = \begin{pmatrix} x_k - x_c \\ y_k - y_c \\ z_k - z_c \end{pmatrix} \times \mathbf{u}_k^0 \tag{4}$$

Here, $\mathbf{I}_{c,k}^0$ is a distance vector between the points on the frames k and c . Moreover, the EBGT mechanism exhibits pure rotation around the x -axis and there is no translation for the components of the system along the x -axis. This implies that the vector moment of \mathbf{u}_k^0 is parallel to the x -axis. The generic form of the moment vector can be obtained by using Eq. (3) in Eq. (4) as follows [19]:

$$\mathbf{r}_{c,k}^0 = \begin{pmatrix} (z_c - z_k) \cdot \sin(\widehat{z_0, z_k}) + (y_k - y_c) \cdot \cos(\widehat{z_0, z_k}) \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} P_{c,k} \\ 0 \\ 0 \end{pmatrix} \tag{5}$$

⁵ All pairs of an EBGT mechanism consist of turning and gear pairs.

We use $P_{c,k}$ as an abbreviation of the first element of the moment vector. Moreover, it can also be expressed as the function of the pitch diameters of the mechanisms, which is further used to represent a coefficient matrix in terms of the speed ratio of gear pairs as described in Sect. 4 [19]. In general, a pitch diameter of a gear indicates the diameter of the pitch circle, which is an imaginary circle that measures the distance between a point on a tooth and its corresponding point on the adjacent tooth. It is used for the characterization of speed ratio of a gear train, which refers to the relative speed of rotation between two rotating components of a gear train. The speed ratio, also known as tooth or gear ratio, is defined as the pitch diameter of the tail component of a gear pair divided by the pitch diameter of its head component. It can be defined as $i_g = d_{g_{tail}}/d_{g_{head}}$ where $g = [g_{tail}, g_{head}]$.

Now, we formalize a screw vector in Isabelle/HOL as:

```
definition screw :: real vec  $\Rightarrow$  real vec  $\Rightarrow$  real vec
where screw u r  $\equiv$  u @v (r  $\times_j$  u)
```

where @_v is the operator for appending vectors, and the operator \times_j is used for the cross-product. Note that, \mathbf{r} refers to the distance vector in 3D space, which will be explicitly modeled in Sect. 4. Next, we formalize the rotation system providing orientation of the unit vectors (Eq. (2)) as follows:

```
definition mat_rot_sys :: real mat  $\Rightarrow$  real vec  $\Rightarrow$  real vec  $\Rightarrow$  bool
where mat_rot_sys  $\Theta$  u uk  $\equiv$  (uk =  $\Theta$  *v u)
```

where Θ denotes the direction cosines matrix and *_v is an operator modeling multiplication between a matrix and a vector. Utilizing this formalization, we verify the general form of angular velocities of each pair (Eq. (3)) as follows:

```
lemma fstvec_form:
assumes dim: u  $\in$  unitvecs and unitz: w = vec_of_list [0, 0, 1]
and rotx_sys: rotx (vec_first (screw u r) 3) w  $\alpha$ 
shows u = vec_of_list [0, sin  $\alpha$ , cos  $\alpha$ ]
```

Here, assumption dim ensures that \mathbf{u} is 3-dimensional unit vector, while unitz asserts that \mathbf{w} is a unit vector of the z -axis. Similarly, rotx_sys ensures that the orientation of \mathbf{u} is about the x -axis with an angle α . The function vec_first accepts a vector and the number 3 and becomes a new vector partitioned according to its first 3 elements (Eq. (1)). Furthermore, the function vec_of_list takes a list and returns it to a vector. Next, we verify the moment vectors for EBGT systems (Eq. (5)) in Isabelle/HOL as follows:

```
lemma sndvec_form:
assumes dimI: r  $\in$  points3D and xdist0: r$0 = 0
and dim: u  $\in$  unitvecs
and unitz: w = vec_of_list [0, 0, 1]
and rotx_sys: rotx (vec_first (screw u r) 3) w  $\alpha$ 
shows vec_last (screw u r) 3 = vec_of_list [-(r$2)*sin  $\theta$ +(r$1)*cos  $\theta$ , 0, 0]
```

Here, `points3D` is a set of 3 dimensional vectors and `xdist0` ensures that there is no translation on the x -axis. `vec_last` is a function that takes a vector v and a number n , and returns a partitioned vector based on the last n elements of the vector v . The verification of the above lemma is based on the `fstvec_form` lemma, alongside some reasoning on vectors, lists and sets. Next, we verify the orthogonality relationship between these vectors as follows:

lemma `rel_fs_orth`:

```
assumes dim: u ∈ unitvecs and dimI: r ∈ points3D
shows vec_first (screw u r) 3 • vec_last (screw u r) 3 = 0
```

The verification of the above lemma is based on the already verified lemmas, such as `fstvec_form` and `sndvec_form`, along with some simplifications on dot product and dimension of the screw vector. Next, the relative angular and linear velocity are defined in a compact form (block/dual vector), called twist $\mathbf{s}_{c,k}^0$ as follows:

$$\mathbf{s}_{c,k}^0 = \mathbf{u}_{c,k}^0 \cdot \dot{q}_k = \begin{pmatrix} \dot{\mathbf{q}}_k^0 \\ \mathbf{I}_{c,k}^0 \times \dot{\mathbf{q}}_k^0 \end{pmatrix} \quad (6)$$

where \dot{q}_k is a scalar velocity variable of a pair (e.g., turning or gear pair) in the mechanism, called twist intensity. The twist intensities assigned to each pair describe the scalar magnitude of the motion between one component of the pair and the other. Similarly, the 3-dimensional vector $\dot{\mathbf{q}}_k^0$ represents the pairs' relative velocities that are also equal the difference between angular velocity of the pair's component (links). Now, we formalize a twist (Eq. (6)) in Isabelle/HOL as:

```
definition twist:: real ⇒ real vec ⇒ real vec ⇒ real vec
where twist q u r ≡ q ·v (screw u r)
```

where \cdot_v is an operator providing multiplication of a scalar and a vector. Similar to the screw, the orthogonality relation between the first and second vector in the twist shall be satisfied, which we verify in Isabelle/HOL as follows:

lemma `twist_fs_orth`:

```
assumes dim: u ∈ unitvecs and dimI: r ∈ points3D
shows vec_first (twist q u r) 3 • vec_last (twist q u r) 3 = 0
```

This lemma is verified using the relationship between screw and twist as well as simplifications on cross and dot product.

3.3 Orthogonality Condition for Kinematic Equations

In order to develop the kinematic equations of an EBGT system, we define the following two orthogonality conditions for angular and linear velocities of pairs, respectively.

$$[C \circ \widehat{\mathbf{u}}_k^0] \cdot \dot{\mathbf{q}}_k^0 = \mathbf{0}_c \quad (7)$$

$$[C \circ \hat{\mathbf{r}}_{c,k}^0] \cdot \hat{\mathbf{q}}_k^0 = \mathbf{0}_c \quad (8)$$

Here, we use the symbol $\hat{}$ to represent a matrix whose elements are vectors. Similarly, \circ represents the Hadamard product, which is used for element-wise multiplication between matrices of the same sizes. $\hat{\mathbf{u}}_k^0$ is a matrix whose elements are unit vectors assigned to each pair of the system, and the matrix has the same dimension as a fundamental cycle matrix C . Moreover, $\hat{\mathbf{r}}_{c,k}^0$ denotes a $c \times k$ matrix whose entries are moment vectors, and $\mathbf{0}_c$ is a c -dimensional vector with zero vector entries. Note that we drop the superscript 0 from the elements of aforementioned matrices for better readability. Since Eqs. (7) and (8) describe the relative velocities and moments of unit vectors, the following conditions are satisfied for every cycle in the cycle basis:

- The sum of twist intensities $\hat{\mathbf{q}}_k = 0$,
- The sum of twists' moments with respect to gear pairs is 0.

Utilizing the above equations, the kinematic equations for the relative velocities of joints can be developed for a given system. For instance, the first condition above provides to obtain twists of gear pairs in terms of twist of turning pairs. Similarly, the solutions of the kinematic equations can be derived in a closed-form by solving the equations for the total number of *DOF* in terms of speed ratios as given in Sect. 4.

The formalization of the orthogonality conditions requires the notion of the Hadamard product, which is formalized in Isabelle/HOL as follows:

```

definition hadamard_prod :: 'a :: semiring_0 mat  $\Rightarrow$  'a vmat  $\Rightarrow$  'a vmat
  where hadamard_prod A B = (let ra = dim_row A; ca = dim_col A in
    if ra = dim_row B  $\wedge$  ca = dim_col B
      then (mat (ra) (ca) ( $\lambda(i,j).$  A $$ (i,j)  $\cdot_v$  B $$ (i,j)))
      else undefined)
    
```

where `dim_row` and `dim_col` are functions that accept a matrix and return the number of rows and columns in the matrix, respectively. Moreover, `'a vmat` and `'a vvec` are the abbreviations for the type synonyms of `'a vec mat` and `'a vec vec`, respectively. Next, we formalize a new operator over reals that provides the multiplication between a matrix (where its components are vectors) and a vector with scalar entries and returns a vector with elements are vectors.

```

definition mult_vmat_vec :: real vmat  $\Rightarrow$  nat  $\Rightarrow$  real vec  $\Rightarrow$  real vvec
  where mult_vmat_vec A n  $\equiv$  vec (dim_row A) ( $\lambda i.$  (vec n ( $\lambda k.$ 
     $\Sigma j < \text{dim\_col } A. (\forall \$ j) * A \$\$ (i, j) \$ k$ )))
    
```

Here, n denotes the size of the vectors in the matrix A . Next, we formalize an m -dimensional vector whose elements are n -dimensional zero vectors, denoted by 0_{vv} , as:

```

definition zero_vvec :: nat  $\Rightarrow$  nat  $\Rightarrow$  'a zero vvec ("0vv")
  where 0vv m n  $\equiv$  vec m ( $\lambda i.$  0v n)
    
```

where 0_v \mathbf{n} describes a n -dimensional zero vector. Using preceding definitions, we now formalize the generic form of the orthogonality condition, which is used to develop Eqs. (7) and (8) for the Bendix wrist mechanism (Sect. 4), as follows:

definition `orth_cond_imp`

where `orth_cond_imp` $A \mathbf{n} v \equiv \text{mult_vmat_vec } A \mathbf{n} v = 0_{vv} \text{ (dim_row } A) \mathbf{n}$

4 Formal Analysis of the Bendix Wrist Mechanism

In this section, we formally analyze a Bendix wrist mechanism (BWM) [19] based on the formalization that we developed in Sect. 3. A BWM is a 3-DOF rotational mechanism which consists of 6 moving links and 6 turning pairs (revolute pairs) and 3 gear pairs as depicted in Fig. 1.

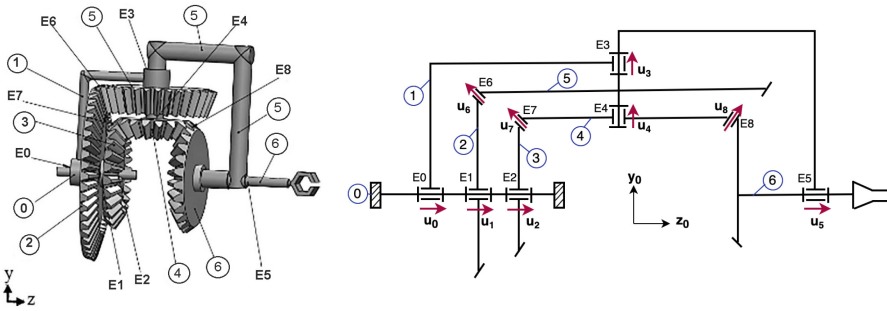


Fig. 1. Bendix Wrist Mechanism [20]

The numbering of the links starts with “0”, which is assigned to the base (or forearm). The mobile links are labeled as {1, 2, 3, 4, 5, 6}, which are the geared wheels and carriers. The number of all pairs in the mechanism is equal to the number of turning pairs labeled {E0, E1, E2, E3, E4, E5} and the number of gear pairs⁶ labeled {E6, E7, E8}. Each pair in the mechanism is attached with the unit vectors to describe the direction of motion. For the above BWM, the unit vectors of turning pairs are considered as

$$\mathbf{u}_0 = \mathbf{u}_1 = \mathbf{u}_2 = \mathbf{u}_5 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \text{ and } \mathbf{u}_3 = \mathbf{u}_4 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \tag{9}$$

In Fig. 2, we present a directed graph of the system with nodes are labeled as mobile links, and edges labeled as pairs. The cycle basis of the system is

⁶ The number of gear pairs is also the number of DOF.

$\{C1, C2, C3\}$. Each edge is represented by a pair of nodes, e.g., $E6 = (5, 6)$ and $E8 = (4, 6)$. Three independent cycles in the system are given as:

$$\begin{aligned}
 C1 &= [E6, -E3, -E0, E1] \\
 C2 &= [E7, E4, -E3, -E0, E2] \\
 C3 &= [E8, -E5, -E4]
 \end{aligned}$$

Here, the negative sign represents the opposite direction between a cycle and an edge in the cycle, e.g., $-E5$ is in cycle $C3$ but their orientations do not coincide. Furthermore, the number of independent cycles is established to be equal to the number of gear pairs in the mechanism.

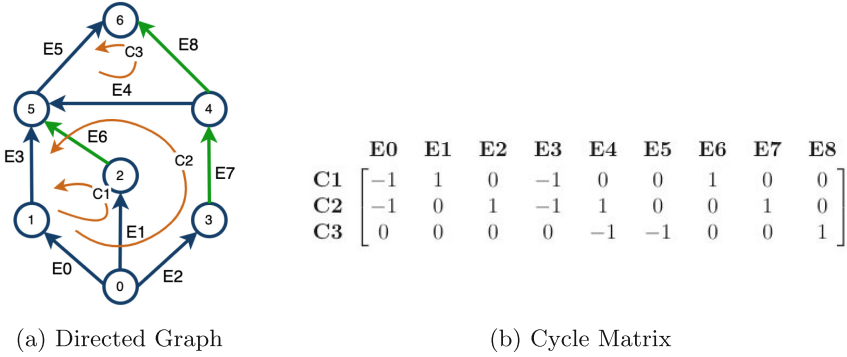


Fig. 2. Directed Graph of BWM and its Corresponding Cycle Matrix

We begin by formally parameterizing the graph components, i.e., nodes, edges, and cycles, which will make the rest of the formalizations easier.

definition `cyc_sys_bwm` where

$$\begin{aligned}
 \text{cyc_sys_bwm} &\equiv \mathcal{N}s = [0, 1, 2, 3, 4, 5, 6] \wedge \\
 \mathcal{E}s &= [(0, 1), (0, 2), (0, 3), (1, 5), (4, 5), (5, 6), (2, 5), (3, 4), (4, 6)] \wedge \\
 \mathcal{L}s &= [[(2, 5), (5, 1), (1, 0), (0, 2)], [(3, 4), (4, 5), (5, 1), (1, 0), (0, 3)], \\
 &\quad [(4, 6), (6, 5), (5, 4)]]
 \end{aligned}$$

Here, nodes and edges are denoted as $\mathcal{N}s$ and $\mathcal{E}s$, and the cycle basis is represented by $\mathcal{L}s$. Note that all formalizations related to cycles are done under the locale `empty_cycle_system` (presented in Sect. 3.1) in order to facilitate the usage of already developed system properties, such as the dimensional and index. We then verify the cycle matrix of BWM (given in Fig. 2b) as follows:

lemma `cycle_matrix_bwm`:

`assumes cyc_sys_bwm`

`shows cycle_matrix $\mathcal{L}s \ \mathcal{E}s = \text{mat_of_rows_list } 9$`

$$\begin{aligned}
 &[[-1, 1, 0, -1, 0, 0, 1, 0, 0], \\
 &[-1, 0, 1, -1, 1, 0, 0, 1, 0], \\
 &[0, 0, 0, 0, -1, -1, 0, 0, 1]]
 \end{aligned}$$

where `mat_of_rows_list` accepts a number indicating the column size of the matrix and a list, which its elements are lists that express each row of the matrix. The verification of the above lemma involves the cycle matrix definition and its properties such as row/column and index properties, negative and positive cycle relationship and reasoning on sets and lists. Next, the first orthogonality condition (Eq. (7)) for BWM is mathematically expressed as:

$$\underbrace{\begin{bmatrix} -\mathbf{u}_0 & \mathbf{u}_1 & \mathbf{0} & -\mathbf{u}_3 & \mathbf{0} & \mathbf{0} & \mathbf{u}_6 & \mathbf{0} & \mathbf{0} \\ \mathbf{u}_0 & \mathbf{0} & \mathbf{u}_2 & -\mathbf{u}_3 & \mathbf{u}_4 & \mathbf{0} & \mathbf{0} & \mathbf{u}_7 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{u}_4 & -\mathbf{u}_5 & \mathbf{0} & \mathbf{0} & \mathbf{u}_8 \end{bmatrix}}_{C \circ \hat{\mathbf{u}}_k^0} \cdot \dot{\mathbf{q}} = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix} \quad (10)$$

where $\dot{\mathbf{q}} = (\dot{q}_0, \dot{q}_1, \dot{q}_2, \dot{q}_3, \dot{q}_4, \dot{q}_5, \dot{q}_6, \dot{q}_7, \dot{q}_8)^T$. Moreover, the characters written in bold represent vectors. To model Eq. (10) for the case of the BWM, we first formalize the $\hat{\mathbf{u}}_k^0$ matrix in Isabelle/HOL as:

```
definition unitvecs_mat :: real vec list  $\Rightarrow$  real vmat
  where unitvecs_mat U  $\equiv$ 
    (let units = [U!0, U!1, U!2, U!3, U!4, U!5, U!6, U!7, U!8]
     in mat_of_rows_list 9 [units, units, units])
```

Now, we formalize the first orthogonality condition (Eq. (7)) as:

```
definition fst_orth_cond :: real vec list  $\Rightarrow$  real vec  $\Rightarrow$  bool
  where fst_orth_cond U q  $\equiv$ 
    orth_cond_imp (hadamard_prod C (unitvecs_mat U)) 3 q
```

where `C` indicates the cycle matrix. The following lemma verifies the Hadamard product of `C` and `unitvecs_mat` utilizing the definitions of the Hadamard product, the matrices as well as their dimensions and index properties.

```
lemma had_mat_fst_bwm:
  assumes cyc_sys_bwm and units_bwm U u6 u7 u8
  shows hadamard_prod C (unitvecs_mat U) = had_mat_fst U
```

Here, the predicate `units_bwm` represents Eq. (9) along with the unknown vectors assigned to gear pairs, i.e., `u6`, `u7`, and `u8`, which are 3-dimensional unit vectors. Furthermore, we define `had_mat_fst` that takes unit vectors `U`, and returns the explicit form of the matrix $C \circ \hat{\mathbf{u}}_k^0$, depicted in Eq. (10). The first orthogonality condition is used to derive the following set of equations in matrix form, which describes the relationship between the gear pairs and the turning pairs twists.

$$\begin{pmatrix} \dot{q}_6 \\ \dot{q}_7 \\ \dot{q}_8 \end{pmatrix} = - \begin{bmatrix} -1 \cdot \mathbf{u}_0 & 1 \cdot \mathbf{u}_1 & \mathbf{0} & -1 \cdot \mathbf{u}_3 & \mathbf{0} & \mathbf{0} \\ -1 \cdot \mathbf{u}_0 & \mathbf{0} & 1 \cdot \mathbf{u}_2 & -1 \cdot \mathbf{u}_3 & 1 \cdot \mathbf{u}_4 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & -1 \cdot \mathbf{u}_4 & -1 \cdot \mathbf{u}_5 \end{bmatrix} \cdot \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{pmatrix} \quad (11)$$

⁷ The superscript “0” is removed from the vectors for better readability.

Here, $\dot{\mathbf{q}}_6, \dot{\mathbf{q}}_7, \dot{\mathbf{q}}_8$ are the first vectors in the twists (relative angular velocity vectors) related to the gear pairs such that $\dot{\mathbf{q}}_6 = \dot{q}_6 \cdot \mathbf{u}_6, \dot{\mathbf{q}}_7 = \dot{q}_7 \cdot \mathbf{u}_7$ and $\dot{\mathbf{q}}_8 = \dot{q}_8 \cdot \mathbf{u}_8$. This relationship is verified in the following lemma.

lemma `rel_gear_turning_pairs`:

```

assumes sys: cyc_sys_bwm and units: units_bwm U u6 u7 u8
assumes twd: dim_vec q = 9
shows fst_orth_cond U q  $\implies$  fst_results_eqs U q

```

where `fst_results_eqs` is formally modeling the set of equations for the relationship between the gear twists and the turning pair twists, described in a compact form in Eq. (11). The verification of the lemma utilizes the verified lemmas `had_mat_fst_bwm` and `fst_orth_eqs`⁸ alongside the definition of `mult_vmat_vec` with reasoning on dimension and index.

Next, the second orthogonality condition (Eq. (8)) for BWM mechanisms can be mathematically expressed as

$$\underbrace{\begin{bmatrix} -\mathbf{r}_{0,0} & \mathbf{r}_{0,1} & \mathbf{0} & -\mathbf{r}_{0,3} & \mathbf{0} & \mathbf{0} & \mathbf{r}_{0,6} & \mathbf{0} & \mathbf{0} \\ -\mathbf{r}_{1,0} & \mathbf{0} & \mathbf{r}_{1,2} & -\mathbf{r}_{1,3} & \mathbf{r}_{1,4} & \mathbf{0} & \mathbf{0} & \mathbf{r}_{1,7} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & -\mathbf{r}_{2,4} & -\mathbf{r}_{2,5} & \mathbf{0} & \mathbf{0} & \mathbf{r}_{2,8} \end{bmatrix}}_{C \circ \hat{\mathbf{r}}_{c,k}^0} \cdot \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \\ \dot{q}_6 \\ \dot{q}_7 \\ \dot{q}_8 \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix} \quad (12)$$

where $\mathbf{r}_{c,k}$ ⁹ are entries of the matrix $\hat{\mathbf{r}}_{c,k}^0$. To formally verify the second orthogonality condition, we need to formalize a few notions, such as the moment matrix and the distance vector. For instance, the moment matrix, $\hat{\mathbf{r}}_{c,k}^0$, is formalized as:

definition `moment_mat`

```

where moment_mat U G T  $\equiv$ 
mat (length G) (length U) ( $\lambda(i,j).$  snd_vecs U G T i j)

```

Here, the function `moment_mat` accepts three real vector lists and returns them in the form of a matrix. Each element of the matrix is obtained through the function `snd_vecs` that maps each index of the moment (second) vector of the screw in the matrix. Similarly, the distance vector (Eq. (4)) is formalized as:

```

fun dist_vec where dist_vec T G i j = (T@G)!j - (G!i)

```

where `dist_vec` is a function that accepts two vector lists as 3-dimensional points in \mathbb{R} and returns them into a difference vector. Next, we verify the relationship between the screw and the moment vector of the screw as follows:

⁸ This lemma and more details about the proof can be found in our Isabelle/HOL script [2].

⁹ For better readability, we remove the superscript “0” from the elements of $\hat{\mathbf{r}}_{c,k}^0$.

lemma `tw_sndvecs_bwm`:

```

assume units_bwm U u6 u7 u8
shows  $\wedge i j. i < \text{length } G \implies j < \text{length } U \implies$ 
  vec_last (screw (U!j) (dist_vec T G i j)) 3 = snd_vecs U T G i j

```

where i and j denote the indices of the lists G and U , respectively. Based on Eq. (4), the distance vector $\mathbf{I}_{c,k} = \mathbf{0}$ when the pair is the gear pair. For the BWM, this equality holds for the gear pairs in the system such that $\mathbf{r}_{0,6} = \mathbf{r}_{1,7} = \mathbf{r}_{2,8} = \mathbf{0}$. In the sequel, we verify these equations, which are further used to generate kinematic equations.

lemma `tw_resultants_bwm`:

```

assume points_param T G and units_bwm U u6 u7 u8
shows "snd_vecs U G T 0 6 = 0_v 3" "snd_vecs U G T 1 7 = 0_v 3"
  "snd_vecs U G T 2 8 = 0_v 3"

```

Here, `points_param` ensures that the elements of the turning pair list T and the gear pair list G are 3-dimensional points represented as vectors, and the size of these lists are 6 and 3, respectively. The verification of above lemma is based on the lemma `tw_sndvecs_bwm`, definitions `snd_vecs`, `dist_vec`, and `screw`, as well as reasoning on cross product, lists and sets. Next, we formalize the second orthogonality condition (Eq. (8)) as follows:

definition `snd_orth_cond`

```

where snd_orth_cond U G T q  $\equiv$ 
  orth_cond_imp (hadamard_prod C (moment_mat U G T)) 3 q

```

Since the moment matrix entries $r_{c,k}$ are functions of $P_{c,k}$ (see Eq. (5)), we can derive a set of equations using the second orthogonality condition as the following matrix form [19]:

$$\begin{bmatrix} -1 \cdot P_{0,0} & 1 \cdot P_{0,1} & 0 & -1 \cdot P_{0,3} & 0 & 0 \\ -1 \cdot P_{1,0} & 0 & 1 \cdot P_{1,2} & -1 \cdot P_{1,3} & 1 \cdot P_{1,4} & 0 \\ 0 & 0 & 0 & 0 & -1 \cdot P_{2,4} & -1 \cdot P_{2,5} \end{bmatrix} \cdot \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad (13)$$

Next, we verify the relation between Eqs. (12) and (13) as follows:

lemma `snd_eqs_ver`:

```

assumes cyc_sys_bwm and units_bwm U u6 u7 u8 and points_param T G
assumes dim_vec q = 9
shows snd_orth_cond U G T q  $\implies$  snd_part_kin_eqs U G T q

```

Here, `snd_part_kin_eqs` represents the equations obtained from Eq. (13). The proof of the above lemma is similar to the verification of the lemma `rel_gear_turning_pairs`. Next, we represent the set of equations (Eq. (13)) in terms of

speed ratios by using the values of scalar $P_{c,k}$'s, which is the first entry of the moment vector of the screw, along each cycle. These values can be given as follows [19]:

$$\begin{aligned}
 \text{Cycle C1} &\longrightarrow P_{0,0} = -0.5 \cdot d_2, P_{0,1} = -0.5 \cdot d_2, P_{0,3} = -0.5 \cdot d_5 \\
 \text{Cycle C2} &\longrightarrow P_{1,0} = P_{1,2} = -0.5 \cdot d_3, P_{1,3} = P_{1,4} = -0.5 \cdot d_4 \\
 \text{Cycle C3} &\longrightarrow P_{2,4} = 0.5 \cdot d_4, P_{2,5} = -0.5 \cdot d_6
 \end{aligned} \tag{14}$$

where d_2, d_3, d_4, d_5 and d_6 represent the pitch diameters, which are labeled the same way as the moving links. Moreover, the speed ratios with respect to each gear (labeled E6, E7 and E8) can be defined using the pitch diameters as:

$$\begin{aligned}
 \text{Gear 0} &\longrightarrow \text{E6} = (2, 5) \longrightarrow \mathbf{i}_0 = d_2/d_5 \\
 \text{Gear 1} &\longrightarrow \text{E7} = (3, 4) \longrightarrow \mathbf{i}_1 = d_3/d_4 \\
 \text{Gear 2} &\longrightarrow \text{E8} = (4, 6) \longrightarrow \mathbf{i}_2 = d_4/d_6
 \end{aligned}$$

Establishing the speed ratios in Eq. (13) using the Eq. (14) as well as some algebraic manipulations, Eq. (13) can be rewritten as:

$$\begin{bmatrix} -\mathbf{i}_0 & \mathbf{i}_0 & 0 & -1 & 0 & 0 \\ -\mathbf{i}_1 & 0 & \mathbf{i}_1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{i}_2 & -1 \end{bmatrix} \cdot \begin{pmatrix} \dot{q}_0 \\ \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{15}$$

Next, we utilize Eq. (15) to find the solution for relative velocities, which is provided in the following vector form:

$$\begin{pmatrix} \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \end{pmatrix} = \begin{pmatrix} -\mathbf{i}_0 \cdot \dot{q}_0 + \mathbf{i}_0 \cdot \dot{q}_1 \\ (-\mathbf{i}_0 + \mathbf{i}_1) \cdot \dot{q}_0 + \mathbf{i}_0 \cdot \dot{q}_1 - \mathbf{i}_1 \cdot \dot{q}_2 \\ (\mathbf{i}_2 \cdot \mathbf{i}_1 - \mathbf{i}_2 \cdot \mathbf{i}_0) \cdot \dot{q}_0 + \mathbf{i}_2 \cdot (\mathbf{i}_0 \cdot \dot{q}_1 - \mathbf{i}_1 \cdot \dot{q}_2) \end{pmatrix}$$

The last step is to formally verify the correctness of the above solution of kinematic equations derived from the second orthogonality condition in Isabelle/HOL as:

theorem `sol_vel_snd_kin`:

```

assumes pitch_diams d2 d3 d4 d5 d6 T G U q
and d4 ≠ 0 d5 ≠ 0 d6 ≠ 0 and dim_vec q = 9
shows rel_vel_sol U G T q ⇒ snd_part_kin_eqs U G T q
    
```

Here, `pitch_diams` refers to the $P_{c,k}$ coefficients (Eq. (14)). The proof process of the above theorem is similar to that of verifying the relative velocities of gear pairs using the first orthogonality condition.

To the best of our knowledge, this is the first formal kinematic analysis of epicyclic bevel gear trains based on both topological matrices and screw theory

in higher-order-logic theorem proving. One of the primary distinctions of our proposed approach for formally analyzing EBGTs is its generic nature. All lemmas are verified for universally quantified variables and functions, allowing for the formal analysis of any EBGT system without the need for individual system modeling, as compared to computer-based simulation methods. For example, the Hadamard product is modeled for generic variables, and specialized for the BWM mechanism (cf. Sect. 4) to verify the lemma `had_mat_fst_bwm`. Similarly, all lemmas are verified for an arbitrary number of components of the EBGTs, which enables the analysis of large and complex systems. For example, we have formalized the cycle graph model for EBGTs systems with any number of links and pairs. Moreover, our work relied on the mathematical analysis provided in the literature, some of which were often ambiguous or lacking in rigorous details, posing significant challenges during the formalization process. However, the use of the Isabelle/HOL theorem prover ensured that every assumption or fact that may have been overlooked in a paper-and-pencil proof, is explicitly provided. The initial focus of our formalization efforts was on providing a generic directed graph model associated with cycle matrices, and combining it with screw theory. To broaden the applicability of the formalization to a variety of scenarios in the modeling and analysis of EBGT systems, we established the locale `cycle_system` based on lists. Moreover, using locale allows flexibility to expand the formalization for potential future analysis and applications. The Isabelle/HOL code for formalization and verification efforts presented in this paper are available at [2].

5 Conclusion

In this work, we proposed to use higher-order-logic theorem proving for the formal kinematic analysis of epicyclic bevel gear train (EBGT) mechanisms. These systems can be analyzed using directed graphs, associated topological matrices, and screw theory concepts. Therefore, we first formally modeled a directed graph including cycle aspects using the locale modules of Isabelle/HOL. We then formalized cycle matrices since these can fully characterize topological properties of the mechanisms, which ease the development and manipulation of kinematic equation in a compact form. We also formally modeled the screw and twist for the relative motion of joints of the mechanisms and used them alongside the cycle matrix to formalize the orthogonality conditions. Finally, we illustrated the effectiveness of our proposed formalization by formally analyzing a Bendix wrist mechanism (BWM), where we formally verified the correctness of the solution of the kinematic equations. As a future work, we plan to extend our analysis through the use of singularity analysis, which enables the detection of changes in the kinematics of the systems [3]. Another future direction would be to explore dynamical aspects of geared mechanisms in order to perform the formal dynamic analysis of more complex physical and engineering systems.

References

1. Affeldt, R., Cohen, C.: Formal foundations of 3D geometry to model robot manipulators. In: *Certified Programs and Proofs*, pp. 30–42. ACM (2017)
2. Aksoy, K.: Formal kinematic analysis of epicyclic bevel gear trains, Isabelle/HOL script (2024). <https://hvg.ece.concordia.ca/code/Isabelle-hol/ebgt.bwm.zip>
3. Amirinezhad, S.V., Donelan, P.: Kinematic singularities of a 3-DOF planar geared robot manipulator. *Adv. Robot Kinematics* **2016**, pp.441–449 (2018)
4. Bagei, C.: Efficient method for the synthesis of compound planetary differential gear trains for multiple speed ratio generation, and constant direction pointing chariots. In: *Applied Mechanisms Conference*, pp. 14–35 (1987)
5. Ball, R.S.: *A Treatise on the Theory of Screws*. Cambridge University Press, Cambridge (1998)
6. Ballarin, C.: Locales: a module system for mathematical theories. *J. Autom. Reason.* **52**(2), 123–153 (2014)
7. Chen, S., Wang, G., Li, X., Zhang, Q., Shi, Z., Guan, Y.: Formalization of camera pose estimation algorithm based on Rodrigues formula. *Formal Aspects Comput.* **32**, 417–437 (2020)
8. Deo, N.: *Graph Theory with Applications to Engineering and Computer Science*. Courier Dover Publications (2017)
9. Edmonds, C., Paulson, L.C.: Formalising Fisher’s inequality: formal linear algebraic proof techniques in combinatorics. In: *Interactive Theorem Proving, LIPICs*, vol. 237, pp. 11:1–11:19 (2022). <https://doi.org/10.4230/LIPICs.ITP.2022.11>
10. Farooq, B., Hasan, O., Iqbal, S.: Formal kinematic analysis of the two-link planar manipulator. In: Groves, L., Sun, J. (eds.) *ICFEM 2013*. LNCS, vol. 8144, pp. 347–362. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41202-8_23
11. Heras, J., Poza, M., Dénès, M., Rideau, L.: Incidence simplicial matrices formalized in Coq/SSReflect. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) *CICM 2011*. LNCS (LNAI), vol. 6824, pp. 30–44. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22673-1_3
12. Huang, Z., Li, Q., Ding, H., Huang, Z., Li, Q., Ding, H.: Basics of screw theory. In: *Theory of Parallel Mechanisms* pp. 1–16. Springer, Dordrecht (2013). https://doi.org/10.1007/978-94-007-4201-7_1
13. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer, Heidelberg (2002). <https://doi.org/10.1007/3-540-45949-9>
14. Norton, R.L.: *Design of Machinery*. McGraw-Hill (2019)
15. Rashid, A., Hasan, O.: Formal verification of robotic cell injection systems up to 4-DOF using HOL Light. *Formal Aspects Comput.* **32**, 229–250 (2020)
16. Recski, A.: *Matroid Theory and its Applications in Electric Network Theory and in Statics*. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-662-22143-3>
17. Shi, Z., Wu, A., Yang, X., Guan, Y., Li, Y., Song, X.: Formal analysis of the kinematic Jacobian in screw theory. *Formal Aspects Comput.* **30**(6), 739–757 (2018). <https://doi.org/10.1007/s00165-018-0468-0>
18. Staicu, S.: Planetary gear train for robotics. In: *International Conference on Mechatronic*, pp. 840–845. IEEE (2005)
19. Talpasanu, I.: A general method for kinematic analysis of robotic wrist mechanisms. *J. Mech. Robot.* **7**(3), 031021 (2015)

20. Talpasanu, I., Simionescu, P.: Kinematic analysis of epicyclic bevel gear trains with matroid method. *J. Mech. Des.* **134**(114501), 1–8 (2012)
21. Tsai, L.W.: The kinematics of spatial robotic bevel-gear trains. *IEEE J. Robot. Autom.* **4**(2), 150–156 (1988)
22. Vullo, V.: *Gears: Volume 1: Geometric and Kinematic Design*, vol. 10. Springer, Cham (2020). <https://doi.org/10.1007/978-3-030-36502-8>
23. Wang, G., Chen, S., Guan, Y., Shi, Z., Li, X., Zhang, J.: Formalization of the inverse kinematics of three-fingered dexterous hand. *J. Logical Algebraic Meth. Program.* **133**, 100861 (2023)
24. Willis, R.: *Principles of Mechanism: Designed for the Use of Students in the Universities, and for Engineering Students Generally*. Cambridge University Press (2010)
25. Wu, A., Shi, Z., Yang, X., Guan, Y., Li, Y., Song, X.: Formalization and analysis of Jacobian matrix in screw theory and its application in kinematic singularity. In: *Intelligent Robots and Systems*, pp. 2835–2842. IEEE (2017)
26. Xie, G., Yang, H., Chen, G.: A framework for formal verification of robot kinematics. *J. Logical Algebraic Meth. Program.* **139**, 100972 (2024)



Deciding the Synthesis Problem for Hybrid Games Through Bisimulation

Catalin Dima¹, Mariem Hammami¹, Youssef Oualhadj^{1,2(✉)},
and Régine Laleau¹

¹ LACL, Université Paris-Est Créteil, 94010 Créteil, France
{catalin.dima,mariem.hammami,regine.aleau}@u-pec.fr

² CNRS, ReLaX, IRL 2000, Siruseri, India
youssef.oualhadj@u-pec.fr

Abstract. Hybrid games are games played on a finite graph endowed with real variables which may model behaviors of discrete controllers of continuous systems. The synthesis problem for hybrid games is decidable for classical objectives (like LTL formulas) when the games are initialized singular, meaning that the slopes of the continuous variables are piecewise constant and variables are reset whenever their slope changes. The known proof adapts the region construction from timed games.

In this paper we show that initialized singular games can be reduced, via a sequence of alternating bisimulations, to timed games, generalizing the known reductions by bisimulation from initialized singular automata to timed automata. Alternating bisimulation is the generalization of bisimulation to games, accommodating a strategy translation lemma by which, when two games are bisimilar and carry the same observations, each strategy in one of the games can be translated to a strategy in the second game such that all the outcomes of the second strategy satisfies the same property that are satisfied by the first strategy. The advantage of the proposed approach is that one may then use realizability tools for timed games to synthesize a winning strategy for a given objective, and then use the strategy translation lemma to obtain a winning strategy in the hybrid game for the same objective.

Keywords: Controller synthesis · Alternating bisimulation · Hybrid games

1 Introduction

In order to describe cyber-physical systems in which discrete and continuous physical processes interact with each other, many mathematical modelling techniques have evolved as a tool. The most common model known so far is the one of hybrid automata [7].

Controller Synthesis and Game Theory. A reactive system interacts with its environment, when this system displays time-sensitive behaviors, it is described

by hybrid games [8]. In practice, some behaviors depend on external factors like weather or temperature. These behaviors are by definition uncontrollable, hence for the system behaves as desired, one needs to find ways to limit the effect of these behaviors. We model this situation through the game theoretic metaphor. We suppose a game played between two players; the first player Pl_1 models the system and is supposed to be controllable, and the second player Pl_2 models the environment and is supposed to be antagonistic.

The specification to ensure is modeled as an objective (*i.e.*, a subset of the possible executions of the system) that Pl_1 has to enforce against any behavior of Pl_2 . This behavior is formalized by the notion of a strategy. Therefore, the *control policy* that we aim to implement is any strategy that ensures the desired specification.

The control problem for hybrid games consists in answering the question of whether there exists a strategy of the controller that ensures a given specification. This problem is undecidable for hybrid games, but decidable for initialized singular games [8], which are games in which the first derivative of each continuous variable is piecewise constant, and whenever this derivative changes, the variable must be reset to some rational value. In this paper we investigate the existence of decidable sub-classes for this problem. The original proof from [8] shows that the region construction for timed automata [1] can be adapted for initialized singular games. Implementing this result requires translating the whole machinery for deciding timed games to the case of initialized singular games, namely the notion of zones which generalize regions, and the controlled predecessor operators on zones. In this article, we show that initialized singular games can be reduced to timed games [6]. Such a reduction would allow a more direct application of existing tools for solving timed games like UppAal TIGA [3]. The reduction is obtained using the notion of *alternating simulation*, which, roughly speaking, is a simulation relation that preserves winning strategies, initially defined for discrete game structures [2] and adapted to concurrent hybrid games in [9].

Contribution. Inspired by the results of [9] over rectangular hybrid automata, we build, from each initialized singular game, a timed game which is alternating bisimilar with the original game. By generalizing [9], the construction passes through intermediate steps in which the original hybrid game is transformed into a stopwatch game, then into an updatable timed game, and finally into a timed game. We also show that each construction comes in pair with an *alternating bisimulation* proving that the resulting game is bisimilar with the previous one. We note that, in some cases, the bisimulation is not a bijection.

Organization. We start by giving the necessary notions and proper definitions concerning hybrid games and turn based games. We then proceed to the construction of a timed turn based game from a given initialized singular turn based game with a bisimulation relation between the two games. This construction is achieved, as mentioned above, through three successive transformations, where in the intermediary steps we build stopwatch games, resp. updatable timed games. Each transformation is accompanied by a relation between the sets of configu-

rations of the two games, which is shown to satisfy the properties of alternating bisimulations. We end with a short section with conclusions and future work.

Due to page limit, the missing material is available in the full version [5].

2 Turn-Based Hybrid Games

Given a set of variables X , the set of simple compact constraints over X is defined by the following grammar:

$$\varphi := x \in I \mid \varphi \wedge \varphi$$

where I is a compact interval with rational bounds and x is a variable in X . For a constraint φ which contains the conjunct $x \in [a, b]$, $\varphi(x)$ denotes the interval $[a, b]$. We consider that each constraint contains a single conjunct of the form $x \in I$ for each variable x .

We extend the framework of Initialized Singular Automata from [9] to the game setting. Informally, an initialized singular game is played between two players, called Pl_1 and Pl_2 . The set of locations is partitioned into two disjoint sets, the first one is controlled by Pl_1 , and the second by Pl_2 . A play is obtained by the following interaction, the players take turns by proposing moves as follows: when a player is in a location that they control. This player proposes an action and a timed delay, hence a new configuration is obtained. The game proceeds from this fresh configuration. By repeating this interaction forever, the two players produce an infinite run for which one of the player wants to satisfy some property (usually called the *objective* of the game).

Definition 1 (Initialized Singular Game). *An initialized singular game (ISR game for short) denoted \mathcal{G}_S of dimension n between two players Pl_1 and Pl_2 is a tuple $\mathcal{G}_S = (L_1, L_2, l_0, X, \text{Act}, \text{Obs}, \text{flow}, E, \text{lbl})$ such that:*

- L_1 is the finite set of locations that belong to Pl_1 and L_2 is the finite set of locations belonging to Pl_2 , with $L_1 \cap L_2 = \emptyset$.
- $l_0 \in L_1$ is the initial location, assumed to belong to Pl_1 .
- X is a set of n real variables.
- Act is a finite set of actions.
- Obs is a finite set of observations.
- $\text{lbl} : (L_1 \cup L_2) \rightarrow \text{Obs}$ is the labeling function that labels each location with an observation.
- $\text{flow} : (L_1 \cup L_2) \times X \rightarrow \mathbb{Q}$ is the value of the derivative which constrains the evolution of each variable in each location.
- $E \subseteq (L_1 \cup L_2) \times \text{Act} \times \text{Cnstr}(X) \times \text{rst}(X) \times (L_1 \cup L_2)$ where $\text{Cnstr}(X)$ is the set of all simple compact constraints over X and $\text{rst}(X) : X \rightarrow \mathbb{Q} \cup \{\perp\}$ is the set of functions that we call reset function. We will denote an edge $e \in E$ as a tuple $e = (l, a, \varphi_e, \text{rst}_e, l')$ such that:
 - $\varphi_e \in \text{Cnstr}(X)$, in order to take the edge e the value of the variables X must satisfy the constraint φ_e .

- rst_e is a reset function of the variables when the edge e is taken, $\text{rst}_e : \mathbf{X} \rightarrow \mathbb{Q} \cup \{\perp\}$, $\text{rst}_e(x) \in \mathbb{Q}$ means that x is reset to a new value and when $\text{rst}_e(x) = \perp$ it means that x is not reset for any $x \in \mathbf{X}$. By abuse of notation, for a variable valuation $v : \mathbf{X} \rightarrow \mathbb{R}$, we also denote $\text{rst}_e(v)$ the valuation defined by

$$\text{rst}_e(v)(x) = \begin{cases} v(x) & \text{when } \text{rst}_e(x) = \perp \\ \text{rst}_e(x) & \text{otherwise} \end{cases}$$

- when $\text{flow}(l, x) \neq \text{flow}(l', x)$ then $\text{rst}_e(x) \neq \perp$. In other words, whenever a variable $x \in \mathbf{X}$ changes its dynamics then its value is reset to $\text{rst}_e(x)$.

The semantics of an ISR game is defined as a transition system

$$T(\mathcal{G}_S) = (Q(\mathcal{G}_S), \text{Mvs}(\mathcal{G}_S), q_0, \delta_S)$$

where $Q(\mathcal{G}_S)$ is the set of configurations, i.e., all the couples $(l, v) \in (L_1 \cup L_2) \times \mathbb{R}^n$ with $q_0 = (l_0, \mathbf{0})$, $\text{Mvs}(\mathcal{G}_S) = \text{Act} \times \mathbb{R}_+$ is the set of moves, and δ_S consists of transitions of the form $(l, v) \xrightarrow{(a, t)} (l', v')$ for which there exists an edge $e = (l, a, \varphi_e, \text{rst}_e, l')$ such that, for each variable $x \in \mathbf{X}$:

- $v(x) + t \cdot \text{flow}(l, x) \in \varphi_e(x)$. We will also use the vectorial notation $v' = v + t \cdot \text{flow}(t, \cdot)$ where $\text{flow}(l, \cdot)$ denotes the vector of variable flows.
- $v'(x) = \text{rst}_e(x)$ whenever $\text{rst}_e(x) \neq \perp$.
- $v'(x) = v(x) + t \cdot \text{flow}(l, x)$ whenever $\text{rst}_e(x) = \perp$.

We denote Q_1 the set of configurations whose location belongs to Pl_1 , i.e. $Q_1(\mathcal{G}_S) = \{(l, v) \in Q(\mathcal{G}_S) \mid l \in L_1\}$ and, similarly, $Q_2(\mathcal{G}_S) = \{(l, v) \in Q(\mathcal{G}_S) \mid l \in L_2\}$. We extend the function lbl over the set of all the configurations $Q_1 \cup Q_2$ as expected. A run ρ in an ISR game is a finite or infinite alternating sequence of configurations and moves, $\rho = q_0 \xrightarrow{(a_1, t_1)} q_1 \xrightarrow{(a_2, t_2)} q_2 \dots$. The set of runs in $T(\mathcal{G}_S)$ is denoted $\text{Runs}(\mathcal{G}_S)$. A history in $T(\mathcal{G}_S)$ is any prefix of a run in $\text{Runs}(\mathcal{G}_S)$, we denote by $\text{Hist}_i(\mathcal{G}_S)$ the set of histories ending in a configuration of Pl_i . A trace is a finite or infinite sequence of observations, and the trace induced by a run ρ is simply $\text{tr}(\rho) = \text{lbl}(q_0) \cdot \text{lbl}(q_1) \cdot \text{lbl}(q_2) \dots$.

Definition 2 (Strategy on $T(\mathcal{G}_S)$). Let $i = 1, 2$. A strategy σ_i of Pl_i is a function that maps finite sequences of configurations ending in a configuration of Pl_i to a move (an action and a time delay).

Given two strategies σ_1 and σ_2 , their outcome $r = q_0 \xrightarrow{m_1} q_1 \xrightarrow{m_2} \dots \in \text{Runs}(\mathcal{G}_S)$ is obtained as follows:

$$q_1 = \delta_S(q_0, \sigma_1(q_0)) \text{ and } m_1 = \sigma_1(q_0)$$

$$q_{i+1} = \begin{cases} \delta_S(q_i, \sigma_1(q_0 \dots q_i)) & \text{if } q_i \in Q_1(\mathcal{G}_S) \text{ and } m_i = \sigma_1(q_0 \dots q_i) \\ \delta_S(q_i, \sigma_2(q_0 \dots q_i)) & \text{if } q_i \in Q_2(\mathcal{G}_S) \text{ and } m_i = \sigma_2(q_0 \dots q_i) \end{cases}$$

We denote $\text{Out}(\sigma_1)$ the set of all runs that are induced by the strategy σ_1 of Pl_1 and some strategy of Pl_2 .

The study of games involves the construction of strategies whose set of outcomes satisfies some desired property.

Semantically, a *property* is a subset of the set of infinite traces. We then say that a strategy σ_1 for Pl_1 satisfies a property P if the set of infinite traces which correspond to outcomes of σ_1 is included in P .

It is known that the problem of synthesizing strategies for properties defined by Linear Temporal Logic formulas is decidable for the case of initialized singular games, as proved in [8] (Theorem 3.1) where an adaptation of the region construction is used. In the rest of the paper, we focus on an alternative approach, namely reducing the synthesis problem for ISR-games to the synthesis problem for timed games by means of successive *alternating bisimulation* reductions. We start by recalling the notion of alternating bisimulation from [8].

2.1 Alternating Simulation for Turn-Based Hybrid Games

In this section we adapt the notion of alternating simulation for the case of (turn-based) ISR-games and a semantic version of the strategy translation property from [2, 8].

Definition 3 (Alternating Simulation on Initialized Compact singular Game). Let \mathcal{G}_S^1 and \mathcal{G}_S^2 be two initialized singular games with the same set of observations Obs , and let $T(\mathcal{G}_S^1) = (Q(\mathcal{G}_S^1), \text{Mvs}(\mathcal{G}_S^1), q_0^1, \delta_S^1)$ and $T(\mathcal{G}_S^2) = (Q(\mathcal{G}_S^2), \text{Mvs}(\mathcal{G}_S^2), q_0^2, \delta_S^2)$ be respectively the induced transition systems.

Let $R \subseteq (Q_1(\mathcal{G}_S^1) \times Q_1(\mathcal{G}_S^2)) \cup (Q_2(\mathcal{G}_S^1) \times Q_2(\mathcal{G}_S^2))$ be a binary relation. The relation R is called a **simulation** if, for any two configurations $p = (l, v) \in Q(\mathcal{G}_S^1)$ and $q = (s, y) \in Q(\mathcal{G}_S^2)$, if $(p, q) \in R$ then:

1. $|\text{bl}^1(p)| = |\text{bl}^2(q)|$
2. If $p \in Q_1(\mathcal{G}_S^1)$ and $q \in Q_1(\mathcal{G}_S^2)$, then for all $(m, t) \in \text{Mvs}(\mathcal{G}_S^1)$ with $p' = (l', v') = \delta_S^1((l, v), (m, t))$ there exists $(m', t') \in \text{Mvs}(\mathcal{G}_S^2)$ and $q' = (s', y') = \delta_S^2((s, y), (m', t'))$ such that $(p', q') \in R$.
3. If $p \in Q_2(\mathcal{G}_S^1)$ and $q \in Q_2(\mathcal{G}_S^2)$, then for all $(m', t') \in \text{Mvs}(\mathcal{G}_S^2)$ and $q' = (s', y') = \delta_S^2(q, (m', t'))$ there exists $(m, t) \in \text{Mvs}(\mathcal{G}_S^1)$ and $p' = (l', v') = \delta_S^1(p, (m, t))$ such that $(p', q') \in R$.

We say that $T(\mathcal{G}_S^2)$ simulates $T(\mathcal{G}_S^1)$ and we denote $T(\mathcal{G}_S^1) \preceq_s T(\mathcal{G}_S^2)$ if there exists a simulation relation $R \subseteq (Q_1(\mathcal{G}_S^1) \times Q_1(\mathcal{G}_S^2)) \cup (Q_2(\mathcal{G}_S^1) \times Q_2(\mathcal{G}_S^2))$ which satisfies all of the above conditions and contains the pair of initial configurations, i.e. $(q_0^1, q_0^2) \in R$. By extension we also denote sometimes $\mathcal{G}_S^1 \preceq_s \mathcal{G}_S^2$.

Moreover, $T(\mathcal{G}_S^1)$ and $T(\mathcal{G}_S^2)$ are bisimilar if there exists simulation relation R witnessing $T(\mathcal{G}_S^1) \preceq_s T(\mathcal{G}_S^2)$ and such that R^{-1} witnesses $T(\mathcal{G}_S^2) \preceq_s T(\mathcal{G}_S^1)$.

Lemma 1 (Simulation Composition). Let \mathcal{G}^1 , \mathcal{G}^2 and \mathcal{G}^3 be three turn based game structures and their respective semantics $T(\mathcal{G}^1), T(\mathcal{G}^2), T(\mathcal{G}^3)$. Assume that α is a simulation between $T(\mathcal{G}^1)$ and $T(\mathcal{G}^2)$, β a simulation between $T(\mathcal{G}^2)$ and $T(\mathcal{G}^3)$ hence $\gamma = \alpha \circ \beta$ is a simulation between $T(\mathcal{G}^1)$ and $T(\mathcal{G}^3)$.

The proof of the above lemma is straightforward and is left as a simple exercise.

Lemma 2. *If $T(\mathcal{G}_S^1) \preceq_s T(\mathcal{G}_S^2)$ then for all σ_1 strategy of Pl_1 in $T(\mathcal{G}_S^1)$, there exists σ'_1 a strategy of Pl_1 in $T(\mathcal{G}_S^2)$ such that*

$$\{\text{tr}(\rho') \mid \rho' \in \text{Out}(\sigma'_1)\} \subseteq \{\text{tr}(\rho) \mid \rho \in \text{Out}(\sigma_1)\}$$

The relevance of this lemma is the following: if we may prove that a game \mathcal{G} is bisimilar with a simpler game \mathcal{G}' and are able to build a strategy σ' for Pl_1 in \mathcal{G}' which satisfies some property, then Lemma 2 allows us to translate σ' into a strategy σ which satisfies the same property. Therefore, if \mathcal{G}' lies in a decidable class of games Γ , we may use the decision algorithm for Γ to synthesize σ' and, hence, provide a methodology for attacking the strategy synthesis problem in the class of games where \mathcal{G} lies.

The plan for the rest of the paper is then to show that ISR games are alternating bisimilar with simpler games in which the strategy synthesis problem is decidable.

3 From Initialized Singular Games to Stopwatch Game

We extend the construction from [9] used for automata to the more general case of turn-based games. The first step is to reduce, through an alternating bisimulation, each initialized singular game to a *stopwatch game*.

Definition 4 (Initialized Stopwatch Game). *An initialized stopwatch game of n -dimension is an initialized singular game in which the flow is either 1 or 0, that is, for any location l and variable x , $\text{flow}(l, x) \in \{0, 1\}$.*

3.1 Transformation

Given an initialized singular game $\mathcal{G}_S = (l_0, L_1, L_2, \mathcal{X}, \text{Act}, \text{Obs}, \text{flow}, E, \text{lbl})$ we construct from \mathcal{G}_S an initialized stopwatch game \mathcal{G}_W , where we change the dynamic to $\text{flow}(l, x) = 1$ when it is not zero, for that we adapt the constraints and the reset functions; we divide the constraints values with the flow of the current location and the reset values with the flow of the successor location. The game is built as follows:

$$\mathcal{G}_W = (l_0, L_1, L_2, \mathcal{X}, \text{Act}, \text{Obs}, \text{flow}_W, E_W, \text{lbl})$$

1. $l_0, L_1, L_2, \text{Act}, \text{Obs}$ and lbl are kept the same as \mathcal{G}_S .
2. $\text{flow}_W : L \times \mathcal{X} \rightarrow \mathbb{Q}^n$ where for all $x \in \mathcal{X}$,

$$\text{flow}_W(l, x) = \begin{cases} 0 & \text{if } \text{flow}(l, x) = 0 \text{ ,} \\ 1 & \text{otherwise .} \end{cases} \tag{1}$$

3. E_W is the set of edges $e_W = (l, a, \varphi_{e_W}, \text{rst}_{e_W}, l')$ such that

- (a) there exists $e = (l, a, \varphi_e, \text{rst}_e, l') \in E$.
 (b) If we denote

$$\varphi_{e_W}(x) = \begin{cases} \frac{\varphi(x)}{\text{flow}(l, x)} & \text{if } \text{flow}(l, x) \neq 0 \text{ ,} \\ \varphi(x) & \text{otherwise .} \end{cases}$$

then the constraint φ_{e_W} is the following conjunction: $\varphi_{e_W} = \bigwedge_{x \in X} \varphi_e(x)$.

- (c) $\text{rst}_{e_W} : X \rightarrow \mathbb{Q}^n \cup \{\perp\}$ where to each $x \in X$ and edge $e = (l, a, \varphi_e, \text{rst}_e, l') \in E$

$$\text{rst}_{e_W}(x) = \begin{cases} \perp & \text{if } \text{rst}(x) = \perp \text{ ,} \\ \frac{\text{rst}(x)}{\text{flow}(l', x)} & \text{if } \text{flow}(l', x) \neq 0 \text{ ,} \\ \text{rst}(x) & \text{if } \text{flow}(l', x) = 0 \text{ .} \end{cases} \quad (2)$$

For the sequel, we denote the semantics of \mathcal{G}_S as $T(\mathcal{G}_S) = (Q(\mathcal{G}_S), \text{Mvs}(\mathcal{G}_S), q_0, \delta)$ and the semantics of \mathcal{G}_W as $T(\mathcal{G}_W) = (Q(\mathcal{G}_W), \text{Mvs}(\mathcal{G}_W), q_0, \delta_W)$.

For the sequel, given a configuration $(l, v) \in Q(\mathcal{G}_S)$, we denote v^* the following variable valuation:

$$\forall x \in X: v^*(x) = \begin{cases} \frac{v(x)}{\text{flow}(l, x)} & \text{if } \text{flow}(l, x) \neq 0 \\ v(x) & \text{otherwise} \end{cases} \quad (3)$$

We define the mapping $\gamma_1 : Q(\mathcal{G}_S) \rightarrow Q(\mathcal{G}_W)$ as follows:

$$\forall q = (l, v) \in Q(\mathcal{G}_S), \gamma_1 : (l, v) \mapsto (l, v^*) .$$

Clearly enough, γ_1^{-1} , the inverse of γ_1 , exists. Therefore, for any $(l, v^*) \in Q(\mathcal{G}_W)$, $\gamma_1^{-1} : (l, v^*) \mapsto (l, v)$ where

$$\forall x \in X: v(x) = \begin{cases} v^*(x) \cdot \text{flow}(l, x) & \text{if } \text{flow}(l, x) \neq 0 \text{ ,} \\ v^*(x) & \text{otherwise .} \end{cases} \quad (4)$$

By construction of $T(\mathcal{G}_W)$ we have $(l, v) \in Q(\mathcal{G}_S)$.

3.2 Bisimulation Between

$T(\mathcal{G}_S)$ and $T(\mathcal{G}_W)$ We now show that the mapping γ_1 defined above is an alternating bisimulation.

Lemma 3. γ_1 witnesses that $T(\mathcal{G}_S) \preceq_s T(\mathcal{G}_W)$.

Proof. Note that, by definition, $q_0^* = \gamma_1(q_0) = q_0$ the initial configurations in the two games $(q_0, q_0^*) \in \gamma_1$ as required.

Let $q = (l, v)$ be a configuration in $Q(\mathcal{G}_S)$, and let $q^* = \gamma_1(q) = (l, v^*)$ be a configuration in $Q(\mathcal{G}_W)$. Since the location l is the same, the observation is preserved thus $\text{lbl}(q) = \text{lbl}(q^*)$, so the first point in Definition 3 holds.

We need to show that the 2nd item in Definition 3 holds, that is, for any two configurations (l, v) in $Q_1(\mathcal{G}_S)$, and (l, v^*) in $Q_1(\mathcal{G}_W)$ with $((l, v), (l, v^*)) \in \gamma_1$ and for any move m in $\text{Mvs}(\mathcal{G}_S)$, there exists a move m^* in $\text{Mvs}(\mathcal{G}_W)$ such that the following holds:

$$(\delta_S((l, v), m), \delta_W((l, v^*), m^*)) \in \gamma_1 . \tag{5}$$

The trick here, is to use the same move m in both $T(\mathcal{G}_S)$ and $T(\mathcal{G}_W)$.

Let (l, v) be a configuration of Pl_1 , and let m be a move, we first prove that $\delta_W((l, v^*), m)$ is well defined. Since $(l', v') = \delta_S((l, v), m)$ with $m = (a, t)$, there exists $e = (l, a, \varphi_e, \text{rst}_e, l')$ in \mathcal{G}_S such that for each $x \in X$,

$$v(x) + t \cdot \text{flow}(l, x) \in \varphi_e(x) \tag{6}$$

and $v' = \text{rst}_e(v)$. By transformation of Sect. 3.1, there exists an edge $e^* = (l, a, \varphi_{e^*}, \text{rst}_{e^*}, l')$ in \mathcal{G}_W . Now, we show that: $v^* + t \cdot \text{flow}_W(l, \cdot)$ satisfies φ_{e^*} . To this end, observe that Identity (6) implies that for all $x \in X$:

$$\begin{aligned} \text{flow}(l, x) \neq 0 &\implies \frac{v(x)}{\text{flow}(l, x)} + t \in \frac{\varphi_e(x)}{\text{flow}(l, x)} \\ &\implies v^*(x) + t \in \varphi_{e^*}(x) \end{aligned}$$

where the second implication is because $\text{flow}(l, x) \neq 0 \implies \text{flow}_W(l, x) = 1$.

When $\text{flow}(l, x) = 0$, $v^*(x) = v(x)$, $\varphi_{e^*}(x) = \varphi_e(x)$ and Identity (6) becomes $v(x) \in \varphi_e(x)$. Hence $v^*(x) \in \varphi_{e^*}(x)$. This implies that $\delta_W(((l, v^*), m)$ is indeed well defined. We still need to show that it is the image of $\delta_S((l, v), m)$ by γ_1 .

Let $(l', v') = \delta_S((l, v), m)$, then we need to study the following cases:

- (1) For $x \in X$ with $\text{flow}(l, x) \neq 0$ and $\text{rst}_e(x) = \perp$:

$$\begin{aligned} \gamma_1((l', v'(x))) &= \left(l', \frac{v'(x)}{\text{flow}(l', x)} \right) = \left(l', \frac{v(x) + t \cdot \text{flow}(l, x)}{\text{flow}(l', x)} \right) \\ &= \left(l', \frac{v(x)}{\text{flow}(l, x)} + t \right) \text{ because } \text{flow}(l', x) = \text{flow}(l, x), \text{ cf. Eq. (3),} \\ &= (l', v^*(x) + t) \text{ because } \text{flow}_W(l', x) = 1, \text{ cf. Eq. (1)} \\ &= \delta_W((l, v^*(x)), m) . \end{aligned}$$

- (2) For all $x \in X$ with $\text{flow}(l, x) = 0$ and $\text{rst}_e(x) = \perp$:

$$\begin{aligned} \gamma_1((l', v'(x))) &= \gamma_1(l', v(x)) \text{ because } \text{flow}(l, x) = 0, \text{ cf. Eq. (1)} \\ &= (l', v^*(x)) = \delta_W((l, v^*(x)), m) \text{ because } \text{flow}_W(l', x) = 0, \text{ cf. Eq. (1).} \end{aligned}$$

- (3) For all $x \in X$ with $\text{flow}(l', x) \neq 0$ and $\text{rst}_e(x) \neq \perp$:

$$\begin{aligned} \gamma_1((l', v'(x))) &= \left(l', \frac{v'(x)}{\text{flow}(l', x)} \right) \\ &= \left(l', \frac{\text{rst}_e(x)}{\text{flow}(l', x)} \right) \text{ because } \text{rst}_e(x) \neq \perp \\ &= (l', \text{rst}_{e^*}(x)) \text{ because } \text{flow}(l', x) = 1, \text{ cf. Eq. (1)} \\ &= \delta_W((l, v^*(x)), m) . \end{aligned}$$

(4) Finally for all $x \in \mathbf{X}$ with $\text{flow}(l, x) = 0$ and $\text{rst}_e(x) \neq \perp$:

$$\gamma_1((l', v'(x))) = (l', v'(x)) = (l', \text{rst}_e(x)) = (l', \text{rst}_{e^*}(x)) = \delta_W((l, v^*(x)), m) .$$

In all of the above cases, we have shown that $\gamma_1(\delta_S((l, v), m)) = \delta_W(\gamma_1((l, v)), m)$, that is, γ_1 preserves the transition relation in both $T(\mathcal{G}_S)$ and $T(\mathcal{G}_W)$.

Now, for proving the 3rd point in Definition 3, Let $q = (l, v)$ be a configuration of Pl_2 , and let m be a move in $\text{Mvs}(\mathcal{G}_W)$, we take the same move in $\text{Mvs}(\mathcal{G}_S)$, we first prove that $\delta_S((l, v), m)$ is well defined in $T(\mathcal{G}_S)$.

Since $(l', v^{*'}) = \delta_W((l, v^*), m)$ with $m = (a, t)$ there exists $e^* = (l, a, \varphi_{e^*}, \text{rst}_{e^*}, l')$ in \mathcal{G}_W such that for each $x \in \mathbf{X}$,

$$v^*(x) + t \cdot \text{flow}_W(l, x) \in \varphi_{e^*}(x) \quad (7)$$

and $v^{*'} = \text{rst}_{e^*}(v)$. By transformation of Sect. 3.1, the existence of e^* in \mathcal{G}_W is the result of the existence of $e = (l, a, \varphi_e, \text{rst}_e, l')$ in \mathcal{G}_S . Now we show that $v + t \cdot \text{flow}(l, \cdot)$ satisfies φ_e . Observe that identity (7) implies that for all $x \in \mathbf{X}$:

$$\begin{aligned} \text{flow}_W(x) \neq 0 &\implies v^*(x) + t \in \varphi_{e^*}(x) \\ &\implies \frac{v(x)}{\text{flow}(l, x)} + t \in \frac{\varphi_e(x)}{\text{flow}(l, x)} \\ &\implies v(x) + t \cdot \text{flow}(l, x) \in \varphi_e(x) \end{aligned}$$

where the second implication is because $\text{flow}_W(x) = 1$ implies $\text{flow}(x) \neq 0$. When $\text{flow}_W(l, x) = 0$, $v(x) = v^*(x)$, $\varphi_e(x) = \varphi_{e^*}(x)$, and identity (7) becomes $v^*(x) \in \varphi_{e^*}(x)$, hence $v(x) \in \varphi_e(x)$. This implies that $q' = \delta_S((l, v), m)$ is indeed well defined with $q' = (l', v')$ and $v' = \text{rst}_e(v + t \cdot \text{flow}(l, \cdot))$. We still need to prove that the image of $\delta_S((l, v), m)$ by γ_1 is $\delta_W((l, v^*), m)$.

For $(l', v^{*'}) = \delta_W((l, v^*), m)$ and we name $(l', v') = \delta_S((l, v), m)$, we have $\forall x \in \mathbf{X}$ with $\text{flow}_W(l, x) \neq 0$ and $\text{rst}_{e^*}(x) = \perp$:

$$\begin{aligned} (l', v^{*'}(x)) &= (l', v^*(x) + t) = \left(l', \frac{v(x)}{\text{flow}(l, x)} + t \right) \\ &= \left(l', \frac{v(x) + t \cdot \text{flow}(l, x)}{\text{flow}(l, x)} \right) = \left(l', \frac{v(x) + t \cdot \text{flow}(l, x)}{\text{flow}(l', x)} \right) \\ &= \left(l', \frac{v'(x)}{\text{flow}(l', x)} \right) = \gamma_1(l', v'(x)) \end{aligned}$$

Consider the case $\text{flow}_W(l, x) = 0$, hence for all $x \in \mathbf{X}$ with $\text{rst}_{e^*}(x) = \perp$:

$$(l', v^{*'}(x)) = (l', v^*(x)) = \gamma_1(l', v(x)) = \gamma_1(l', v'(x))$$

Now for all $x \in X$ with $\text{flow}_W(l', x) \neq 0$ and $\text{rst}_{e^*}(x) \neq \perp$:

$$\begin{aligned} (l', v^{*'}(x)) &= (l', \text{rst}_{e^*}(x)) = \left(l', \frac{\text{rst}_{e^*}(x)}{\text{flow}(l', x)} \right) \\ &= \left(l', \frac{v(x) + t \cdot \text{flow}(l, x)}{\text{flow}(l, x)} \right) = \left(l', \frac{v(x) + t \cdot \text{flow}(l, x)}{\text{flow}(l', x)} \right) \\ &= \left(l', \frac{v'(x)}{\text{flow}(l', x)} \right) = \gamma_1(l', v'(x)) \end{aligned}$$

Finally, for all $x \in X$ with $\text{flow}_W(l', x) = 0$ and $\text{rst}_{e^*}(x) \neq \perp$:

$$(l', v^{*'}(x)) = (l', \text{rst}_{e^*}(x)) = (l', \text{rst}_e(x)) = (l', v'(x)) = \gamma_1(l', v'(x))$$

Therefore, $\gamma_1(\delta_S((l, v), m)) = \delta_W(\gamma_1((l, v), m))$. This ends the proof of this lemma. \square

Lemma 4. γ^{-1} witnesses that $T(\mathcal{G}_W) \preceq_s T(\mathcal{G}_S)$.

Proof. Note that, by definition, $q_0 = \gamma_1^{-1}(q_0^*)$ the initial configurations in the two games as required.

Let $q^* = (l, v^*)$ be a configuration in $Q(\mathcal{G}_W)$ and let $q = (l, v) = \gamma_1^{-1}(q^*)$ be a configuration in $Q(\mathcal{G}_S)$. Since the location l is the same, the observation is preserved thus $\text{lbl}(q) = \text{lbl}(q^*)$, so the first point in Definition 3 holds.

We now show that γ_1^{-1} satisfies the 2nd point in Definition 3, for all $q^* = (l, v^*) \in Q(\mathcal{G}_W)$ and $q = (l, v) = \gamma_1^{-1}(q^*) \in Q(\mathcal{G}_S)$. Assume that q^* a configuration of Pl_1 , for all $m = (a, t) \in \text{Mvs}(T(\mathcal{G}_W))$ and $q^{*'} = (l', v^{*'}) = \delta_W(q^*, m)$ in $T(\mathcal{G}_W)$, we take the same move m in $T(\mathcal{G}_S)$, we first prove that $\delta_S(q, m)$ is well defined in $T(\mathcal{G}_S)$, let us call it q' . Afterwards we prove that $q' = \gamma_1^{-1}(q^{*'})$. We have for each $x \in X$,

$$v^{*'}(x) = v^*(x) + t \cdot \text{flow}_W(l, x) \in \varphi_{e^*}(x)$$

And $v^{*'} = \text{rst}_{e^*}(v^*)$ for the edge $e^* = (l, a, \varphi_{e^*}(x), \text{rst}_{e^*}(x), l')$, e^* is obtained from the edge $e = (l, a, \varphi_e(x), \text{rst}_e(x), l')$ in \mathcal{G}_S by transformation of Sect. 3.1. Now we show that $v + \text{flow}(l, \cdot) \in \varphi_e$. For all $x \in X$ and $\text{flow}(l, x) \neq 0$ we have

$$\begin{aligned} v^*(x) + t \cdot \text{flow}_W(l, x) &\in \varphi_{e^*}(x) \\ \implies v^*(x) \cdot \text{flow}(l, x) + t \cdot \text{flow}_W(l, x) \cdot \text{flow}(l, x) &\in \varphi_{e^*}(x) \cdot \text{flow}(l, x) \\ \implies v(x) + t \cdot \text{flow}(l, x) &\in \varphi_e(x) \cdot \text{flow}(l, x) \\ \implies v(x) + t \cdot \text{flow}(l, x) &\in \varphi_e(x) \end{aligned}$$

For all $x \in X$ and $\text{flow}(l, x) = 0$, we obtain $v^*(x) = v(x)$ and $\varphi_{e^*}(x) = \varphi_e(x)$. It follows that $v(x) \in \varphi_e(x)$. This implies that indeed $q' = \delta_S(q, m) = (l', v')$ with $v' = \text{rst}_e(v + t \cdot \text{flow}(l, \cdot))$ is well defined.

Now let us prove $q' = \gamma_1^{-1}(q^{*'})$. For all $x \in X$:

(1) For $\text{flow}(l, x) \neq 0$ and $\text{rst}_{e^*} = \perp$ we have:

$$\begin{aligned}\gamma_1^{-1}(l', v^{*'}) &= (l', v^{*'}(x) \cdot \text{flow}(l', x)) = (l', (v^*(x) + t \cdot \text{flow}_W(l, x)) \cdot \text{flow}(l', x)) \\ &= (l', (v^*(x) + t) \cdot \text{flow}(l, x)) = (l', (v^*(x) \cdot \text{flow}(l, x) + t \cdot \text{flow}(l, x))) \\ &= (l', v + t \cdot \text{flow}(l, x)) = (l', v'(x))\end{aligned}$$

(2) For $\text{rst}_{e^*} \neq \perp$ and $\text{flow}(l', x) \neq 0$ we have:

$$\begin{aligned}\gamma_1^{-1}(l', v^{*'}) &= (l', v^{*'}(x) \cdot \text{flow}(l', x)) = (l', \text{rst}_{e^*}(x) \cdot \text{flow}(l', x)) \\ &= (l', \text{rst}_e(x)) = (l', v'(x))\end{aligned}$$

(3) For $\text{rst}_{e^*} \neq \perp$ and $\text{flow}(l', x) = 0$ we have:

$$\gamma_1^{-1}(l', v^{*'}) = (l', v^{*'}(x)) = (l', \text{rst}_{e^*}(x)) = (l', \text{rst}_e(x)) = (l', v'(x))$$

We conclude that $(l', v') = \gamma_1^{-1}(l', v^{*'})$.

Now, for proving the 3rd point in Definition 3: let q^* be a configuration of Pl_2 , and let m be a move in $\text{Mvs}(\mathcal{G}_S)$ and $q' = (l', v') = \delta_S((l, v), m)$ in $T(\mathcal{G}_S)$, we take the same move in $\text{Mvs}(\mathcal{G}_W)$, we first prove that $q^{*'} = \delta_W((l, v^*), m)$ is well defined in $T(\mathcal{G}_W)$, and then we prove that $\gamma_1^{-1}(q^{*'}) = (q')$.

Since $q' = (l', v') = \delta_S((l, v), m)$ then there exists an edge $e = (l, a, \varphi_e, \text{rst}_e, l')$. By transformation there exists an edge in \mathcal{G}_W $e^* = (l, a, \varphi_{e^*}, \text{rst}_{e^*}, l')$.

For each $x \in \mathbf{X}$ with $\text{flow}(l, x) \neq 0$, we have:

$$\begin{aligned}v(x) + t \cdot \text{flow}(l, x) &\in \varphi_e(x) \\ \implies v^*(x) \cdot \text{flow}(l, x) + t \cdot \text{flow}(l, x) &\in \frac{\varphi_e(x)}{\text{flow}(l, x)} \cdot \text{flow}(l, x) \\ \implies v^*(x) + t &\in \varphi_{e^*}(x) \\ \implies v^*(x) + t \cdot \text{flow}_W(l, x) &\in \varphi_{e^*}(x)\end{aligned}\tag{8}$$

Notice that for $\text{flow}(l, x) = 0$, we obtain $\text{flow}_W(l, x) = 0$, $v^*(x) = v(x)$, $\varphi_e(x) = \varphi_{e^*}(x)$, hence $v^*(x) \in \varphi_{e^*}(x)$. Let $v^{*'} = \text{rst}_{e^*}(v^* + t \cdot \text{flow}_W(l, \cdot))$. We have proved that $q^{*'} = (l, v^{*'}) = \delta_W((l, v^*), m)$ is well defined.

Now we prove that $\gamma_1^{-1}(l', v^{*'}) = (l', v')$. For each $x \in \mathbf{X}$ consider the case $\text{flow}(l', x) \neq 0$ and $\text{rst}_{e^*} = \perp$,

$$\begin{aligned}\gamma_1^{-1}(l', v^{*'}(x)) &= (l', v^{*'}(x) \cdot \text{flow}(l', x)) = (l', (v^*(x) + t \cdot \text{flow}_W(l, x)) \cdot \text{flow}(l', x)) \\ &= (l', v^*(x) \cdot \text{flow}(l', x) + t \cdot \text{flow}(l', x)) \\ &= (l', v^*(x) \cdot \text{flow}(l, x) + t \cdot \text{flow}(l, x)) \\ &= (l', v(x) + t \cdot \text{flow}(l, x)) = (l', v'(x))\end{aligned}$$

Now for each $x \in \mathbf{X}$ $\text{flow}(l', x) = 0$ and $\text{rst}_{e^*} = \perp$,

$$\gamma_1^{-1}(l', v^{*'}(x)) = (l', v^{*'}(x)) = (l', v^*(x)) = (l', v(x)) = (l', v'(x))$$

Same for the case for each $x \in X$ with $\text{rst}_{e^*}(x) \neq \perp$ and $\text{flow}(l', x) = 0$,

$$\gamma_1^{-1}(l', v^{*'}(x)) = (l', v^{*'}(x)) = (l', \text{rst}_{e^*}(x)) = (l', \text{rst}_e(x)) = (l', v'(x))$$

Finally consider the case for each $x \in X$ with $\text{rst}_{e^*}(x) \neq \perp$ and $\text{flow}(l', x) \neq 0$,

$$\begin{aligned} \gamma_1^{-1}(l', v^{*'}(x)) &= (l', \text{rst}_{e^*}(x) \cdot \text{flow}(l', x)) = \left(l', \frac{\text{rst}_e(x)}{\text{flow}(l', x)} \cdot \text{flow}(l', x) \right) \\ &= (l', \text{rst}_e(x)) = (l', v'(x)) \end{aligned}$$

This ends the proof. \square

4 From Initialized Stopwatch to Updatable Timed Games

The next step is to transform each initialized stopwatch game into a game where the dynamics of each variable is never zero, by eliminating all the flows of value zero. The games obtained by this transformation are called *updatable timed games*.

Definition 5 (Updatable Timed Game). *An updatable timed game \mathcal{G}_U of n -dimension is an initialized singular game in which all variables are clocks, i.e. $\text{flow}(l, x) = 1$ for any location l and any variable $x \in X$.*

Note that these structures generalize timed games by allowing updates with any rational value, similarly with updatable timed automata from [4].

4.1 Transformation

Given \mathcal{G}_W an initialized stopwatch game with n variables (whose semantics is denoted $T(\mathcal{G}_W)$ in the sequel) we construct an updatable timed game in two steps. We first transform \mathcal{G}_W into another stopwatch game $\overline{\mathcal{G}_W}$ in which the locations carry some extra information about resets. Then this stopwatch game is transformed into an updatable timed game.

The need for the intermediate stopwatch game comes from the fact that each stopwatch in the original game is simulated by a clock in the resulting game. But clocks are always incremented when time pass, so when some stopwatch in a location l in which the stopwatch's flow is 0 must be encoded with a clock, we need to reset this clock on any transition leaving l . The value to which this clock must be updated depends on the sequence of transitions through which l has been reached. Along a finite run ρ ending with the edge e this reset value it is the update to which the stopwatch was reset on the latest edge e' when the stopwatch's flow changed from 1 to 0. These bits of memory are modeled by the extra information encoded into locations of the new stopwatch game.

The formalization of the first transformation is the following:

$$\overline{\mathcal{G}_W} = (\overline{l}_0, \overline{L}_1, \overline{L}_2, X, \text{Act}, \text{Obs}, \text{flow}, \overline{E}, |\text{bl}|)$$

1. $\bar{l}_0 = (l_0, f_{l_0})$ where $f_{l_0} : X \rightarrow \{0\}$.
2. $\bar{L}_1 = L_1 \times F$ where F is the set that consists of functions $f : X \rightarrow K_\perp$, where K is the set of all constants used in \mathcal{G}_W and $K_\perp = K \cup \{\perp\}$.
3. $\bar{L}_2 = L_2 \times F$.
4. lbl is extended over $\bar{L}_1 \cup \bar{L}_2$ as expected.
5. \bar{E} is the set of edges $\bar{e} = (\bar{l}_1, a, \varphi, \text{rst}, \bar{l}_2)$ where $\bar{l}_1 = (l_1, f_{l_1}), \bar{l}_2 = (l_2, f_{l_2})$ and there exists an edge $e \in E$ with $e = (l_1, a, \varphi, \text{rst}, l_2)$ and for all $x \in X$:

$$f_{l_2}(x) = \begin{cases} \perp & \text{if } \text{flow}(l_2, x) = 1 \text{ ,} \\ \text{rst}(x) & \text{if } \text{flow}(l_2, x) = 0 \wedge \text{rst}(x) \neq \perp \text{ ,} \\ f_{l_1}(x) & \text{if } \text{flow}(l_2, x) = 0 \wedge \text{rst}(x) = \perp \text{ .} \end{cases} \quad (9)$$

The semantics of $\overline{\mathcal{G}_W}$ that we call $T(\overline{\mathcal{G}_W})$ is as follows

$$T(\overline{\mathcal{G}_W}) = (Q(\overline{\mathcal{G}_W}), \text{Mvs}(\overline{\mathcal{G}_W}), \bar{q}_0, \bar{\delta}_W)$$

Now we construct \mathcal{G}_U an updatable timed game from $\overline{\mathcal{G}_W}$ in the next step.

Step 2:

$$\mathcal{G}_U = (\bar{l}_0, \bar{L}_1, \bar{L}_2, X, \text{Act}, \text{Obs}, E_U, \text{lbl})$$

1. $\bar{l}_0, \bar{L}_1, \bar{L}_2, \text{Obs}, \text{Act}, \text{lbl}$ are all same as in $\overline{\mathcal{G}_W}$.
2. E_U is the set of edges $e_U = (\bar{l}_1, a, \varphi, \text{rst}_U, \bar{l}_2)$ with $\text{rst}_U : X \rightarrow \mathbb{Q}^n \cup \{\perp\}$ such that for all $\bar{e} \in \bar{E}$ with $\bar{e} = (\bar{l}_1, a, \varphi, \text{rst}, \bar{l}_2)$ we have for all $x \in X$,

$$\text{rst}_U(x) = \begin{cases} f_l(x) & \text{if } f_l(x) \neq \perp \text{ ,} \\ \text{rst}(x) & \text{otherwise .} \end{cases} \quad (10)$$

Note that if we have a reset of a variable x with $\text{flow}(l, x) = 0$, then $\text{rst}_U(x) = \text{rst}(x) = f_l(x)$. Note that E_U is same as \bar{E} with different reset function $\text{rst}_U(X)$ on each edge.

The transition system of the updatable timed game \mathcal{G}_U , that we call $T(\mathcal{G}_U)$ is as follows

$$T(\mathcal{G}_U) = (Q(\mathcal{G}_U), \text{Mvs}(\mathcal{G}_U), q_{0_U}, \delta_U)$$

We define the relation $\beta_1 \subseteq Q(\mathcal{G}_W) \times Q(\overline{\mathcal{G}_W})$ as follows:

$$\beta_1 = \{((l, v), (\bar{l}, \bar{v})) \mid (l, v) \in Q(\mathcal{G}_W), \bar{l} = (l, f_l) \text{ where } f_l : X \rightarrow K_\perp \text{ and } \bar{v} = v\}$$

And its inverse $\beta_1^{-1} \subseteq Q(\overline{\mathcal{G}_W}) \times Q(\mathcal{G}_W)$,

$$\beta_1^{-1} = \{((\bar{l}, \bar{v}), (l, v)) \mid (\bar{l}, \bar{v}) \in Q(\overline{\mathcal{G}_W}), ((l, v), (\bar{l}, \bar{v})) \in \beta_1\}$$

4.2 Bisimulation Between $T(\mathcal{G}_W)$ and $T(\overline{\mathcal{G}_W})$

In this section we prove that β_1 is a bisimulation relation.

Lemma 5. β_1 witnesses $T(\mathcal{G}_W) \preceq_s T(\overline{\mathcal{G}_W})$.

Proof. Note that, already by definition $(\bar{l}_0, v_0) = ((l_0, f_{l_0}), v_0)$ and (l_0, v_0) are the initial configurations in the two games, hence $((l_0, v_0), (\bar{l}_0, v_0)) \in \beta_1$ as required.

Let $(l, v) \in Q(\mathcal{G}_W)$ and $(\bar{l}, v) = ((l, f_l), v) \in Q(\overline{\mathcal{G}_W})$ with $((l, v), (\bar{l}, v)) \in \beta_1$. Since the location l is the same, the observation is preserved thus $\text{lbl}(l, v) = \text{lbl}(\bar{l}, v)$. So the first point of the Definition 3 holds.

Let (l, v) be a configuration of Pl_1 , let $m = (a, t) \in \text{Mvs}(\mathcal{G}_W)$ and $(l', v') = \delta_W((l, v), (a, t))$ in $T(\mathcal{G}_W)$. To prove point 2 in Definition 3, we take the same m in $\text{Mvs}(\overline{\mathcal{G}_W})$ and prove that $\bar{\delta}_W((\bar{l}, v), (a, t)) = (\bar{l}', v')$ is well defined, and then $((l', v'), (\bar{l}', v')) \in \beta_1$. $\delta_W((l, v), (a, t)) = (l', v')$ implies that there exists an edge $e = (l, a, \varphi_e, \text{rst}_e, l')$ in \mathcal{G}_W , hence, by construction 9 there exists $\bar{e} = ((l, f_l), a, \varphi_e, \text{rst}_e, (l', f_{l'}))$ in $\overline{\mathcal{G}_W}$. This implies that $\bar{\delta}_W((\bar{l}, v), (a, t)) = (\bar{l}', v')$ with $\bar{l}' = (l', f_{l'})$ is well defined in $T(\overline{\mathcal{G}_W})$. It follows that $((l', v'), (\bar{l}', v')) \in \beta_1$ since v' is the same for the two configurations and $f_{l'}$ is well defined.

For proving the third point of the Definition 3, let (l, v) be a configuration of Pl_2 , and $m = (a, t) \in \text{Mvs}(\overline{\mathcal{G}_W})$ with $\bar{\delta}(((l, f_l), v), (a, t)) = ((l', f_{l'}), v')$ in $\overline{\mathcal{G}_W}$. Similarly to the above proof, we take the same move m in $\text{Mvs}(\mathcal{G}_W)$, then prove that $\delta_W((l, v), (a, t)) = (l', v')$ is well defined and $((l', v'), (\bar{l}', v')) \in \beta_1$. This follows by noting that the existence of transition $\bar{\delta}(((l, f_l), v), (a, t)) = ((l', f_{l'}), v')$ implies there exists an edge $\bar{e} = ((l, f_l), a, \varphi_e, \text{rst}_e, (l', f_{l'}))$ in $\overline{\mathcal{G}_W}$. By construction 9, \bar{e} is obtained from $e = (l, a, \varphi_e, \text{rst}_e, l')$ in \mathcal{G}_W . Note that rst_e is preserved by the construction. Hence $(l', v') = \delta_W((l, v), (a, t))$ with $v' = \text{rst}_e(v + t \cdot \text{flow}(l, \cdot))$ is well defined, which ends the proof. \square

Lemma 6. β_1^{-1} witnesses $T(\overline{\mathcal{G}_W}) \preceq_s T(\mathcal{G}_W)$.

The proof of this lemma is similar to the proof of Lemma 5

We define the mapping $\beta_2 : Q(\overline{\mathcal{G}_W}) \rightarrow Q(\mathcal{G}_U)$ as follows:

$$\forall (\bar{l}, v) = ((l, f_l), v) \in Q(\overline{\mathcal{G}_W}), \beta_2 : ((l, f_l), v) \mapsto ((l, f_l), v) .$$

Lemma 7. β_2 is the bisimulation between $T(\overline{\mathcal{G}_W})$ and $T(\mathcal{G}_U)$.

Proof. β_2 is the identity function hence it is clear that it is a bisimulation. \square

Now let $\beta : Q(\mathcal{G}_W) \rightarrow Q(\mathcal{G}_U)$ as $\beta = \beta_1 \circ \beta_2$. And it's inverse $\beta^{-1} : Q(\mathcal{G}_U) \rightarrow Q(\mathcal{G}_W)$ as $\beta^{-1} = \beta_2^{-1} \circ \beta_1^{-1}$. Lemmas 7 and 1 directly imply the following:

Corollary 1. β is a bisimulation between $T(\mathcal{G}_W)$ and $T(\mathcal{G}_U)$.

5 From Updatable Timed Game to Timed Game

Definition 6 (Timed Turn Based Game). A timed turn based game \mathcal{G}_T of n -dimension is an updatable timed game in which clocks can only be reset to 0.

In the sequel, in an edge $e = (l, a, \varphi_e, r, l')$ of a timed automaton, we consider that $r \subseteq X$ denotes the set of clocks which are reset along this edge.

5.1 Transformation

Given an updatable game $\mathcal{G}_U = (l_0, L_1, L_2, \mathsf{X}, \text{Act}, \text{Obs}, \text{E}, \text{lbl})$ whose semantics is denoted $T(\mathcal{G}_U) = (Q(\mathcal{G}_U), \text{Mvs}(\mathcal{G}_U), q_0, \delta_U)$ we now construct the timed game \mathcal{G}_T as:

$$\mathcal{G}_T = (l_0^t, L_1^t, L_2^t, \mathsf{X}, \text{Act}, \text{Obs}, \text{E}^t, \text{lbl})$$

where

1. $l_0^t = (l_0, g_{l_0})$ where $g_{l_0} : \mathsf{X} \rightarrow \{0\}$.
2. $L_1^t = L_1 \times F_t$ where F_t is the set of functions $g : \mathsf{X} \rightarrow K_U$, where K_U is the set of constants used in the definition of \mathcal{G}_U .
3. $L_2^t = L_2 \times F_t$.
4. lbl is extended over $L_1^t \cup L_2^t$ as expected.
5. E^t is the set of edges $e^t = (l_1^t, a, \varphi_{e^t}, r, l_2^t)$ where $l_1^t = (l_1, g_{l_1})$ and $l_2^t = (l_2, g_{l_2})$ such that there exists an edge $e \in \text{E}$ with $e = (l_1, a, \varphi_e, \text{rst}_e, l_2)$ and for all $x \in \mathsf{X}$:

$$\varphi_{e^t} = \bigwedge_{x \in \mathsf{X}} \varphi_e(x) \text{ with } \varphi_{e^t}(x) = \varphi_e(x) - g_{l_1}(x) \quad (11)$$

$$g_{l_2}(x) = \begin{cases} \text{rst}(x) & \text{if } \text{rst}(x) \neq \perp \wedge x \in r \\ g_{l_1}(x) & \text{if } \text{rst}(x) = \perp \end{cases} \quad (12)$$

$$x \in r \iff \text{rst}_e(x) \neq \perp \quad (13)$$

The transition system of the constructed timed game \mathcal{G}_T is denoted:

$$T(\mathcal{G}_T) = (Q(\mathcal{G}_T), \text{Mvs}(\mathcal{G}_T), q_0^t, \delta_T)$$

We define the relation $\gamma_2 \subseteq Q(\mathcal{G}_U) \times Q(\mathcal{G}_T)$ as follows:

$$\gamma_2 = \{((l, v), (l^t, v^t)) \mid (l, v) \in Q(\mathcal{G}_U), l^t = (l, g_l) \text{ where } g_l : \mathsf{X} \rightarrow K_U \\ \text{and for all } x \in \mathsf{X} v^t(x) = v(x) - g_l(x)\}$$

And its inverse $\gamma_2^{-1} \subseteq Q(\mathcal{G}_T) \times Q(\mathcal{G}_U)$,

$$\gamma_2^{-1} = \{((l^t, v^t), (l, v)) \mid (l^t, v^t) = ((l, g_l), v^t) \in Q(\mathcal{G}_T), \\ \forall x \in \mathsf{X} v(x) = v^t(x) + g_l(x)\}$$

5.2 Bisimulation Between $T(\mathcal{G}_U)$ and $T(\mathcal{G}_T)$

We now show that the mapping γ_2 defined above is an alternating bisimulation.

Lemma 8. γ_2 witnesses that $T(\mathcal{G}_U) \preceq_s T(\mathcal{G}_T)$.

Proof (Sketch). Recall that $(l_0^t, v_0^t) = ((l_0, g_{l_0}), v_0^t)$ is the initial configuration in $T(\mathcal{G}_T)$ and (l_0, v_0) is the initial configuration in $T(\mathcal{G}_U)$, where $v_0 = v_0^t$, hence $((l_0, v_0), (l_0^t, v_0^t)) \in \gamma_2$ as required. Now we need to prove the three points in Definition 3.

Let $(l, v) \in Q(\mathcal{G}_U)$ and $(l^t, v^t) = ((l, g_l), v^t) \in Q(\mathcal{G}_T)$ with $((l, v), (l^t, v^t)) \in \gamma_2$. Since the location l is the same, the observation is preserved thus $\text{lbl}(l, v) = \text{lbl}(l^t, v^t)$, so the first point in Definition 3 holds.

Next we show that γ_2 satisfies the 2nd point in Definition 3. Meaning for $(l, v) \in Q_1(\mathcal{G}_U)$, $m = (a, t) \in \text{Mvs}(\mathcal{G}_U)$ and $\delta_U((l, v), m) = (l', v')$, we apply the same move in $T(\mathcal{G}_T)$ and we first prove that $\delta_T((l^t, v^t), m) = (l^{t'}, v^{t'})$ is well defined, and then $((l', v'), (l^{t'}, v^{t'})) \in \gamma_2$.

To show that γ_2 satisfies the third point in Definition 3, for each (l, v) in $Q_2(\mathcal{G}_U)$, each $m = (a, t)$ in $\text{Mvs}(\mathcal{G}_T)$, and each $\delta_T(((l, g_l), v^t), m) = ((l', g_{l'}), v^{t'})$, we apply the same move in $T(\mathcal{G}_U)$. This requires showing that $\delta_U((l, v), m) = (l', v')$ is well defined, and then that $((l', v'), ((l', g_{l'}), v^{t'})) \in \gamma_2$. The details for this part of the proof are available in the extended version [5]. \square

Lemma 9. γ_2^{-1} witnesses that $T(\mathcal{G}_T) \preceq_s T(\mathcal{G}_U)$.

Proof (Sketch). Recall that the initial configuration in $T(\mathcal{G}_T)$ is $(l_0^t, v_0^t) = ((l_0, g_{l_0}), v_0^t)$ and (l_0, v_0) is the initial configuration in $T(\mathcal{G}_U)$, where $v_0^t = v_0$ because $g_{l_0} = 0$, hence $((l_0^t, v_0^t), (l_0, v_0)) \in \gamma_2^{-1}$ as required. The scheme is as above, now we prove step by step the three points in Definition 3.

Notice that the location l is the same for (l^t, v^t) and (l, v) , hence the observation is preserved. It follows that $\text{lbl}(l^t, v^t) = \text{lbl}(l, v)$, therefore the first point in Definition 3 holds. Then we prove that γ_2^{-1} satisfies the second point in Definition 3. Meaning that for $(l^t, v^t) \in Q_1(\mathcal{G}_T)$, a configuration of Pl_1 , let $m = (a, t) \in \text{Mvs}(\mathcal{G}_T)$ and $\delta_T(((l, g_l), v^t), m) = ((l', g_{l'}), v^{t'})$. By applying the same move in $T(\mathcal{G}_U)$, we prove first that $\delta_U((l, v), m) = (l', v')$ is well defined, and later on we prove $((l', g_{l'}), v^{t'}), (l', v') \in \gamma_2^{-1}$.

To show that γ_2^{-1} satisfies the third point in Definition 3, for each configuration $((l, g_l), v^t)$ in $Q_2(\mathcal{G}_T)$, each $m = (a, t)$ in $\text{Mvs}(T(\mathcal{G}_U))$, and each $\delta_U((l, v), m) = (l', v')$, we take the same move m in $\text{Mvs}(T(\mathcal{G}_T))$. We then prove that $\delta_T(((l, g_l), v^t), m) = ((l', g_{l'}), v^{t'})$ is well defined and that $((l', g_{l'}), v^{t'}), (l', v') \in \gamma_2^{-1}$. Again, the details of this part of the proof is available in the extended version [5]. \square

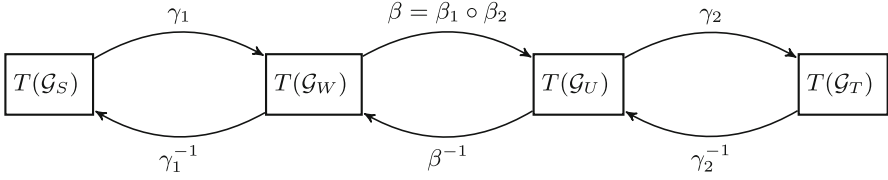


Fig. 1. Summary of the chain of transformations

6 Conclusion

We have shown that each initialized singular game is bisimilar with a timed game in the following way. First, given an initialized singular game $T(\mathcal{G}_S)$ we construct an initialized stopwatch game $T(\mathcal{G}_W)$. We give the relation γ_1 and prove that it is a bisimulation relation as defined in Definition 3. Then given an initialized stopwatch game $T(\mathcal{G}_W)$, we construct an updatable timed game $T(\mathcal{G}_U)$. We construct the relation β and prove that it is a bisimulation relation and finally given an updatable timed game $T(\mathcal{G}_U)$, we construct a timed game $T(\mathcal{G}_T)$. We give the relation γ_2 and prove that it is a bisimulation relation. These constructions are summarized in Fig. 1.

The decidability of the synthesis problem for LTL specifications and initialized singular games can then be seen as a corollary of this result.

As future research directions, we plan to extend this result by carefully adapting the results for larger sub-classes of initialized rectangular hybrid automata from [9] and hence weakening the constraint that the derivative of each variable must be piece-wise constant. We also plan to apply our methodology to real-life case-studies.

Acknowledgments. This work was funded by grant ANR-17-CE25-0005 the **DISCONT** Project funded by the *Agence Nationale de la Recherche (ANR)*.

The first and third authors are supported by grant ANR-20-CE25-0012 the **MAVeriQ** Project funded by the *Agence Nationale de la Recherche (ANR)*.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
2. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) *CONCUR 1998*. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055622>
3. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: time for playing games! In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_14
4. Bouyer, P., Dufourd, C., Fleury, E., Petit, A.: Updatable timed automata. *Theor. Comput. Sci.* **321**(2-3), 291–345 (2004). <https://doi.org/10.1016/j.tcs.2004.04.003>

5. Dima, C., Hammami, M., Oualhadj, Y., Laleau, R.: Deciding the synthesis problem for hybrid games through bisimulation (2024). <https://arxiv.org/abs/2409.05498>
6. Faella, M., Torre, S.L., Murano, A.: Dense real-time games. In: LICS'02, pp. 167–176. IEEE Computer Society (2002). <https://doi.org/10.1109/LICS.2002.1029826>
7. Henzinger, T.A.: The theory of hybrid automata. In: LICS'96, pp. 278–292. IEEE Computer Society (1996). <https://doi.org/10.1109/LICS.1996.561342>
8. Henzinger, T.A., Horowitz, B., Majumdar, R.: Rectangular hybrid games. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 320–335. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48320-9_23
9. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? *J. Comput. Syst. Sci.* **57**(1), 94–124 (1998). <https://doi.org/10.1006/jcss.1998.1581>



Formal Analysis of FreeRTOS Scheduler on ARM Cortex-M4 Cores

Chen-Kai Lin^(✉) and Bow-Yaw Wang

Academia Sinica, Taipei, Taiwan
kai.zsv@gmail.com, bywang@iis.sinica.edu.tw

Abstract. FreeRTOS is a real-time kernel with configurable scheduling policies. It is one of the most popular real-time kernel for embedded devices due to its portability and configurability. We formally analyze the FreeRTOS scheduler on ARM Cortex-M4 processor in this work. Concretely, we build a formal model for the FreeRTOS ARM Cortex-M4 port and apply model checking to find errors in our models for FreeRTOS example applications. Intriguingly, several errors are found in our application models under different scheduling policies. In order to confirm our findings, we modify application programs distributed by FreeRTOS and reproduce assertion failures on the STM32F429I-DISC1 board.

Keywords: model checking · real-time kernel · ARMv7-M architecture

1 Introduction

FreeRTOS is a popular open-sourced real-time kernel [3]. It offers multi-tasking on uni-processor embedded devices. Through multi-tasking, applications can be divided into several simpler tasks sharing processor time. Multi-tasking on the other hand can induce undesirable phenomena such as deadlocks or starvation. It is crucial to prevent such errors in deployment. Multi-tasking errors nevertheless are notoriously evasive. Due to complex interleaving among tasks, a very limited number of system behaviors can be tested. Software testing often fails to detect multi-tasking errors.

Model checking is a formal technique to analyze properties about systems [8]. Behaviors of the system under verification are first specified in a formal model. Model checkers then verify the model automatically with formal properties provided by users. Different from testing tools, model checkers search model behaviors exhaustively. If a deviant behavior is found, it is reported to verifiers. If no deviance can be found, all model behaviors conform to the specified property. The formal model is thus verified.

In this paper, we develop formal models for the FreeRTOS scheduler on ARM Cortex-M4 processors and analyze its properties with the SPIN model checker. We build formal models for the ARM Cortex-M4 interrupt handling mechanism based on the reference manual. Optimizing mechanisms such as tail chaining are implemented in our models. On top of our hardware model, we also build

formal models for the FreeRTOS scheduler, thread-safe data structures, and its applications by examining source codes of the FreeRTOS ARM Cortex-M4 port. Particularly, all three FreeRTOS scheduling policies are specified in our behavior models.

With our behavior models for FreeRTOS, it remains to identify formal properties to check. Such properties however can be tricky to find. For formal analysis, high-level informal properties such as deadlocks or starvation need to be specified concretely. In complex systems like FreeRTOS, high-level properties are often asserted with caveats to preclude minor or unrealistic errors. It can be very tedious to specify caveats formally. Moreover, one can not be sure of these caveats without FreeRTOS developers' help. Different developers can also have different views on properties and caveats. Formal properties specified by verifiers themselves can be contrived or even incorrect.

We solve the property specification problem by verifying example applications in the FreeRTOS distribution. In order to highlight FreeRTOS features, developers provide a number of example applications. Most example applications contain assertions to specify expected behaviors during execution. No assertion failure should be observed on any multi-tasking execution. In order to verify whether all assertions hold in all possible executions, we verify them with the SPIN model checker. Intriguingly, the model checker reports errors on several example application models.

Assertion errors found in formal analysis need not imply assertion failures in real execution. In order to confirm our findings, we modify FreeRTOS example applications to reproduce error traces found by the SPIN model checker. If assertion errors in formal analysis are genuine, we should observe assertion failures on real hardware. Using the STM32F429I-DISC1 board from STMicroelectronics, we successfully reproduce assertion failures in our experiments. We use the remote GDB debugger to confirm failures at intended assertions. All assertion failures require delicate interactions among tasks, the FreeRTOS scheduler, and the ARM Cortex-M4 interrupt mechanism.

This paper is organized as follows. Section 2 introduces the SPIN model checker. Section 3 presents analysis methodology. Section 4 briefly introduces our behaviors models for the FreeRTOS scheduler and ARM Cortex-M4 port. Section 5 defines properties of our models. Section 6 reports verification results and discussion. Section 7 gives related works. We conclude this work in Sect. 8.

2 Background

Model checking is an automatic formal verification technique. In model checking, behaviors of systems under verification are specified as formal models. Properties about systems are also formalized by logical properties about formal models. Given a formal model and a logical property, a model checker automatically verifies the property against the model through mathematical reasoning. If the model is verified, the property holds in the model mathematically. If the model is not verified, the model checker returns a trace to witness the error.

SPIN is a model checker designed for analyzing communicating concurrent processes [16]. It supports the PROMELA (PROcess MEta LAnguage) language to specify system behaviors. A PROMELA model consists of a set of processes. A process contains a sequence of *commands*. Commands must be *enabled* before executed. Enabled commands in different processes are executed interleavably. That is, exactly one enabled command is executed at any time. If several commands from different processes are enabled, one of the enabled commands is executed non-deterministically. If there is no enabled command, it is a *deadlock*. The PROMELA language allows verifiers to specify assertions in processes. An *assertion* command contains a Boolean expression and is always enabled. When an assertion command is executed, it evaluates its Boolean expression. If the value is false, it is an assertion error. Recall that enabled commands are executed non-deterministically. Non-deterministic executions result in different *traces*. An assertion error may appear in some traces but not all of them.

Since traces correspond system behaviors, a deadlock or an assertion error in any trace represent undesirable behaviors. We therefore would like to check if deadlocks or assertion errors occur among all traces. The SPIN model checker systematically explores all traces with sophisticated algorithms. If a deadlock or an assertion error occurs in any trace, SPIN will find the *error trace* and report it as a witness. If SPIN does not find any deadlock or assertion error after exploring all traces, the model is verified.

In addition to assertions, SPIN allows verifiers to specify properties with Linear Temporal Logic (LTL) formulas. Particularly, we will use the LTL formula $\Box\Diamond Loc$ where *Loc* denotes a process location. A trace satisfies $\Box\Diamond Loc$ if it visits the process location *Loc* infinitely many times. A process satisfies $\Box\Diamond Loc$ if all its traces satisfy $\Box\Diamond Loc$. The formula $\Box\Diamond Loc$ specifies that a process is free of starvation. For instance, let *Loc* be the location where a process finishes its job. A trace satisfies $\Box\Diamond Loc$ if the process finishes its job infinitely many times and hence without starvation.

3 Methodology Overview

To support different architectures, the FreeRTOS scheduler contains both architecture-dependent and -independent codes. Scheduling policies should be independent of underlying architectures. They provide abstract programming models for applications. Their implementations however necessarily depend on interrupt mechanisms in underlying architectures. Concretely, the FreeRTOS scheduler is called during periodic and sporadic interrupts in the ARM Cortex-M4 port. For exhaustive analysis, it is essential to consider all possible interrupt sequences. Generating such interrupt sequences for testing is infeasible. A more effective technique is required.

We develop a PROMELA model for the interrupt mechanism on ARM Cortex-M4 processors. Behaviors of ARM Cortex-M4 processors are carefully formalized in our model. Importantly, we model interrupts through non-determinism. Non-deterministic interrupts allow us to explore interrupt sequences unattainable by testing.

On top of our formal model for the ARM Cortex-M4 interrupt mechanism, we then specify a PROMELA model for the architecture-independent codes in the FreeRTOS scheduler. All three FreeRTOS scheduling policies are specified in our model. Our formal model for the FreeRTOS scheduler on ARM Cortex-M4 processors enables extensive analysis on task synchronization. We moreover build formal models for thread-safe data structures widely used by FreeRTOS applications like queues and locks.

With formal models, we proceed to verify properties about the FreeRTOS scheduler. Although abstract properties such as the absence of deadlock and starvation are easily said, they are not precise enough for formal analysis. Additionally, properties are unlikely to be satisfied without provisions. Without FreeRTOS developers' inputs, contrived or even misleading properties can be verified meaninglessly.

We address this problem by verifying FreeRTOS example applications. Similar to most open-sourced projects, FreeRTOS provides example applications to illustrate its features. These applications contain assertions to specify expected behaviors. These assertions are properties written by FreeRTOS developers. No assertion failure should be observed under all circumstances. In order to verify assertions in FreeRTOS example applications, we build their formal models and check if an assertion error might occur. Intriguingly, several assertion errors were found in our analysis.

It is important to recall that formal models are different from real hardware and software by definition. Assertion errors found on the models do not necessarily correspond to assertion failures on real systems. In order to validate our findings, we examine the error traces found by the SPIN model checker and reproduce them on the STM32F429I-DISC1 board. We use the remote debugger GDB to confirm assertion failures on the ARM Cortex-M4 board. Errors found by our formal analysis are successfully realized on the development board. These assertion failures require intricate interrupt events. They are unlikely to be found by traditional testing.

4 FreeRTOS Scheduler Model

Our goal is to develop PROMELA models for the ARM Cortex-M4 interrupt mechanism, the FreeRTOS scheduler, thread-safe data structures, and example applications. An application has a number of tasks to be executed by the processor. When an interrupt is triggered, its interrupt handler will be executed by the processor. We therefore say a task or an interrupt handler is an *execution unit*. In our model, an execution unit is formalized as a PROMELA process. Commands in a process thus specify the sequential computation of the execution unit.

4.1 Execution Units

In PROMELA, an enabled command is executed non-deterministically among all such commands in all processes. Execution units however need to be scheduled by

our formal scheduler model before execution. To this end, we define the global variable EP (for Executing Process) and assign each execution unit a unique identification number. Every command in execution units are added with the condition $EP == id$. The FreeRTOS scheduler model in turn assigns the variable EP to elect next command.

Task. Typical FreeRTOS tasks loop forever and never terminate. Their models are PROMELA processes with infinite loops.

ARM Cortex-M4 Interrupt Handler. For ARM Cortex-M4 processors, an interrupt is triggered when it is set to the *pending* state. When an interrupt is pending, the processor decides whether the current execution should be interrupted. If the pending interrupt is unmasked *and* has a priority over the current execution, the current execution is interrupted by the pending interrupt and the corresponding interrupt handler is executed.

ARM Cortex-M4 processors optimizes nested interrupts. When returning from an interrupt handler, the processor checks if there is any pending interrupt with a priority over the interrupted execution. If so, the processor proceeds to the pending interrupt rather than the interrupted execution. This optimization is called *tail chaining*.

Similar to task models, our interrupt models are PROMELA processes with infinite loops. Commands within the loops are guarded with the EP variable. An iteration in the interrupt model is executed one time whenever the interrupt conditions (pending, masking, and priority) are satisfied. Tail chaining is also specified in our models.

4.2 FreeRTOS Scheduler

The FreeRTOS scheduler provides three scheduling policies. In *cooperative scheduling*, a running task has to yield the processor explicitly. In *preemptive scheduling without time slicing*, a running task can be preempted by tasks with higher priorities. Finally, a task can moreover be interrupted by using up its time slice in *preemptive scheduling with time slicing*. The next execution task is elected by the policy.

In the FreeRTOS Cortex-M4 port, scheduling policies are implemented via two interrupt handlers: PendSV and SysTick. The interrupt handler for the software interrupt *PendSV* elects the next execution task. The PendSV interrupt is triggered whenever a task needs to be rescheduled in all scheduling policies.

The *SysTick* interrupt implements the time slicing policy. It is triggered by a hardware clock periodically. If time slicing is enabled, the SysTick interrupt handler in turn triggers the PendSV interrupt. When the SysTick interrupt handler finishes, the PendSV interrupt handler will be executed directly by tail chaining.

Our PROMELA model follows the FreeRTOS Cortex-M4 port to specify the scheduler in PendSV and SysTick interrupt models. Since the PROMELA language is timeless, our model cannot trigger the SysTick interrupt periodically.

Effectively, the SysTick interrupt is triggered arbitrarily in our formalization. The abstraction ensures that all SysTick interrupt sequences in real world are subsumed in our model. If there is any failure among all real interrupt sequences, it will be exposed in our model. However, not all interrupt sequences in our model are real. An error found in the model can be spurious. It has to be validated by corresponding failures in real hardware.

In cooperative scheduling, a task calls the FreeRTOS yield function to release the processor. The yield function triggers the PendSV interrupt to elect a task in the PendSV interrupt handler. It is straightforward to define the yield function in our model.

4.3 Task Synchronization

In addition to task scheduling, the FreeRTOS scheduler also provides basic functions for task synchronization. More concretely, a task can be *delayed* for a specified duration; it can also be *suspended* indefinitely. When a task is delayed or suspended, it is moved to a delay or suspended queue respectively and hence cannot be scheduled for execution. Delayed tasks can be rescheduled when their delay duration expire. Suspended tasks can be rescheduled when they are resumed by the running task.

In the FreeRTOS Cortex-M4 port, basic task synchronization functions are implemented by the PendSV and SysTick interrupt handlers as well. The SysTick interrupt handler checks if any task in the delay queue has expired its duration periodically. If so, the interrupt handler removes such tasks from the delay queue. For suspended tasks, they are removed from the suspended queue when they are resumed by the running task.

If preemptive scheduling is disabled, the running task continues its execution until it yields the processor. If preemptive scheduling is enabled, the PendSV interrupt is triggered when the tasks removed from the delay or suspended queues have a priority over the running task. The FreeRTOS scheduler elects a task with the highest priority for execution. A previously delayed or suspended task will continue its execution; and the running task will be preempted if it does not have the priority.

To handle the degenerative scenario where all tasks are delayed or suspended, FreeRTOS adds an *idle* task. The idle task has the lowest priority and cannot be delayed nor suspended. Instead, it can be configured to yield the processor or not. If the idle task should yield, it yields the processor to the next scheduled task immediately. Otherwise, the idle task loops until it is interrupted. The idle task is also formalized in our model.

Our model for task synchronization mostly follows the FreeRTOS Cortex-M4 port. One major difference between our model and the port is timelessness. Since delay duration cannot be formalized explicitly, we formalize delay duration by a `Tick` counter and a set of `Delay` counters. When a task model is delayed, the corresponding `Delay` counter is set. When the SysTick interrupt handler model is executed, it increases the `Tick` counter by one. The `Delay` counter of a task model expires if it equals the `Tick` counter. When their counters expire, the

SysTick interrupt handler model removes such tasks from the delay queue. All counters are reset when a task model is added to or removed from the delay queue to prevent counter overflow.

4.4 Thread-Safe Data Structures

In addition to task synchronization, FreeRTOS provides thread-safe data structures for message passing and advanced synchronization among tasks. A thread-safe structure consists of its data and a *waiting* task queue. A task modifies a thread-safe structure if its data are ready. Otherwise, the task is *blocked*. When a task is blocked, it is moved to the waiting task queue of the thread-safe structure for specified duration. Different from task synchronization, tasks can be unblocked when its duration expires or data become ready. It is a failure if the duration of a blocked task expires before the data are ready.

FreeRTOS implements thread-safe queues for message passing. A thread-safe queue contains a bounded buffer as its data. The capacity of the buffer is specified by programmers. The buffer is ready when it is neither full for sending nor empty for receiving message. When a task modifies an unready buffer, it is blocked for the specified duration. Consider a sender is blocked by sending a message to a full buffer. When a message is removed from the buffer, the sender will be unblocked immediately. If the buffer remains full when the duration expires, the sender receives a failure. The duration can be zero. It is a failure if the thread-safe queue is currently not ready.

FreeRTOS also offers thread-safe locks. A thread-safe lock uses a counter as its data. Programmers can initialize the counter. If the counter is a positive integer, it means the lock is ready to be taken by tasks. Otherwise, the lock is not ready. Tasks are blocked for a specified duration if they try to take an unready lock. The duration can be zero or a positive integer. It is a failure if the lock remains unready when the duration expires. No task will be blocked when it gives the lock.

Thread-safe structures are widely used in FreeRTOS applications. They are specified in our models. Thanks to our ARM Cortex-M4 interrupt and FreeRTOS scheduler models, our thread-safe structure models mostly follow the FreeRTOS Cortex-M4 port.

4.5 Example Applications

Tasks behave very differently in different scheduling policies. Consider a task which never yields. In preemptive scheduling, such a task can be preempted by other tasks with sufficient priorities. Tasks with higher priorities can still be scheduled for execution. On the other hand, a never-yielding task is never preempted in cooperative scheduling. Since other tasks will not execute, no progress is made. To ensure progress, FreeRTOS developers make low-priority or never-yielding tasks actively yield the processor when preemption is disabled. Such intricacies can be a burden to programmers.

To help programmers develop their applications smoothly, FreeRTOS provides example applications in its distribution. Particularly, mutexes and semaphores are used for task synchronization. Thread-safe queues are also found in applications for message passing. We have constructed formal models for eight example applications such as *PollQ*, *Semtest*, *BlockQ*, *QPeek*, *Dynamic*, *Countsem*, *Recmutex*, and *GenQTest*. These applications illustrate task synchronization or thread-safe structures in FreeRTOS. They undoubtedly are relevant to the formal analysis of FreeRTOS scheduler in this work.

5 Formal Properties

Formal analysis of FreeRTOS scheduler demands formal properties. Such properties nevertheless are not always obvious to verifiers. High-level properties are too obscure to be formalized. Local properties are often contrived with little relevance. To avoid such pitfalls, we verify assertions annotated by developers in FreeRTOS distribution.

Assertions in FreeRTOS scheduler detect errors in tasks and thread-safe structures at runtime. Task assertions fail when the scheduler resumes a non-suspended task. Assertions in thread-safe structures fail when a mutex is used as a semaphore, or a mutex inherits the priority of a wrong task. Other assertions check the capacity of thread-safe buffers and task priorities. Those assertions are verified in our formal analysis.

Not all assertions are similar however. To organize our presentation, we classify assertions in example applications into two categories. Intuitively, an assertion specifies a safety property if it indicates that a bad event should never happen; an assertion specifies a liveness property if it indicates that a good event should always happen.

5.1 Safety

It is straightforward to specify safety properties with assertions. Programmers only need to write a Boolean expression deemed to be true in an assertion. In FreeRTOS example applications, the following safety properties are found:

- (S0) If a task is delayed for synchronization with other tasks, other tasks must finish before the delay duration expires.
- (S1) If a task is blocked by thread-safe data, data must be ready when it is unblocked.
- (S2) If a task expects a thread-safe data to be ready, the data must be ready.
- (S3) Messages received through a thread-safe queue must preserve their order.
- (S4) Mutexes and binary semaphores must ensure mutual exclusion of critical sections.
- (S5) If a thread-safe lock is taken once, it must be given eventually.
- (S6) A low-priority task must inherit priorities when its mutex was taken by tasks with higher priorities and recover its priority after releasing the mutex.

Property (S0) checks if task synchronization is used properly. Property (S1) checks if thread-safe data are implemented correctly. Property (S2) is a special case of property (S1) where the block duration is zero. Property (S3) checks messages are delivered in order by thread-safe queues. Properties (S4) and (S5) check mutexes and semaphores are implemented correctly. Finally, property (S6) checks whether priority inheritance is implemented correctly.

Not all properties are needed in every application. Table 1 shows the safety properties specified in the eight example applications.

Table 1. Properties in FreeRTOS Applications

	Safety						Liveness
	S0	S1	S2	S3	S4	S5	
<i>PollQ</i>	✓		✓	✓			✓
<i>Semtest</i>	✓	✓	✓		✓		✓
<i>BlockQ</i>		✓	✓	✓			✓
<i>QPeek</i>		✓	✓	✓			✓
<i>Dynamic</i>	✓		✓	✓	✓		✓
<i>Countsem</i>			✓			✓	✓
<i>Recmutex</i>		✓			✓	✓	✓
<i>GenQTest</i>		✓	✓	✓	✓	✓	✓

5.2 Liveness

If a task does nothing, no bad event can happen. The task thus satisfies all safety properties. To avoid such vacuous safety, liveness properties are specified. FreeRTOS developers in fact write assertions to ensure tasks are making progress. Concretely, a task maintains a counter which

is incremented when a job is finished. The counter is checked by a monitor task periodically. An assertion failure occurs if the counter remains unchanged between checks.

Monitor tasks do not contribute to the computation. They also interfere with the scheduler. Accuracy of checking liveness properties by monitor tasks may be in doubt. Instead of checking progress by monitor tasks, we specify liveness properties by LTL formulas and get rid of monitor tasks in our models. Our analysis hence removes unnecessary disruption in the scheduler. It is more precise than testing liveness with monitors.

Let $Loc_{SysTick}$ be the location triggering the SysTick interrupt and Loc_i the location where task model i finishes its job for $1 \leq i \leq n$. Consider the LTL formula:

$$\Box\Diamond Loc_{SysTick} \rightarrow (\Box\Diamond Loc_1 \wedge \Box\Diamond Loc_2 \wedge \dots \wedge \Box\Diamond Loc_n)$$

Informally, the formula states that all tasks finish their jobs infinitely many times if the SysTick interrupt is triggered infinitely many times. In our formal models, SysTick interrupts represent the progression of time. If the LTL formula is satisfied in our models, it means that all task models must finish their jobs infinitely often as time progresses. No task can stop making progress indefinitely. The liveness property is required for all FreeRTOS application models in Table 1.

Table 2. Verification Time in Seconds

Applications	Cooperative Scheduling		Preemptive w/o Time Slicing		Preemptive w/ Time Slicing	
	Safety	Liveness	Safety	Liveness	Safety	Liveness
<i>PollQ</i>	1.6	33.6	3.7	111.0	5.2	154.0
<i>Semtest</i>	0.1	✗	58.0	✗	517.0	✗
<i>QPeek*</i>	< 0.1	1.3	< 0.1	1.1	< 0.1	✗
<i>Recmutex*</i>	7.2	305.0	5.6	267.0	39.5	✗
<i>Countsem*</i>	< 0.1	0.4	< 0.1	✗	3.2	✗
<i>GenQTest*</i>	0.9	98.3	< 0.1	✗	197.0	✗
<i>Dynamic*</i>	5.8	131.0	0.1	✗	S0	✗
<i>BlockQ*</i>	1.3	210.0	2.2	549.0	S1	✗

* Some tasks in the application actively yield the processor when preemption is disabled.

6 Verification Results

For each scheduling policy, we use the model checker SPIN to verify properties shown in Table 1. The model checker first verifies safety properties in an application model. After checking safety properties, the liveness property is verified on the application model. In our experiments, we use SPIN 6.5.2 on an Ubuntu 20.04 server with two 3.2 GHz octa-core CPUs and 512 GB RAM.

Table 2 gives the verification results for safety and liveness properties in eight applications under three scheduling policies. If all safety properties in an application are satisfied, the verification time (in seconds) is shown. If not, the failed property is shown with a cross mark in the table. For the liveness property, verification time is shown if an application satisfies the property. Otherwise, a cross mark is shown.

6.1 Analysis of Safety Properties

Almost all applications satisfy their safety properties. SPIN finishes the verification with at most 40 GB of memory in 10 min. For failed safety properties, the model checker also reports error traces with 10 GB memory in 1 min.

Under preemptive scheduling with time slicing, SPIN reports that *Dynamic* and *BlockQ* violate safety properties (S0) and (S1) respectively. In error traces reported by SPIN, we find that a task may not execute even though it is scheduled by the FreeRTOS scheduler. To see how it happens, consider the SysTick interrupt is triggered while the PendSV interrupt handler is running. Since both interrupts have the same priority, the SysTick interrupt is pending until the PendSV interrupt handler finishes. Recall that the PendSV interrupt handler calls the scheduler to elect a task for execution. Let us call the elected task as the *victim*. The victim task is scheduled to execute after the exception returns. However, the SysTick interrupt is still pending. Due to tail chaining, the SysTick interrupt handler will execute before the victim task. In the time slicing policy,

the SysTick interrupt handler will trigger another PendSV interrupt to schedule a task. The scheduler incorrectly believes the victim task has used up its time slice and chooses another task for execution. In error traces, the victim task repeatedly misses its time slice and hence cannot prepare the thread-safe queue shared with another blocked task. When the blocked task expires its duration, the shared thread-safe queue is still not ready. *Dynamic* and *BlockQ* hence violate safety properties (S0) and (S1) respectively.

Although assertion errors are found in our formal analysis, they are not necessarily failures in reality. It is important to recall that our application models are not FreeRTOS example applications. During model construction, abstraction and simplification are indispensable for effective formal analysis. For instance, the SysTick interrupt is not triggered periodically in our timeless formal models. It is therefore important to reproduce assertion failures in real hardware. To this end, we install the FreeRTOS V10.5.1 on the STM32F429I-DISC1 board with an ARM Cortex-M4 processor and modify FreeRTOS example applications to reproduce SPIN error traces on the board. An on-board LED will flash with high frequency if an assertion failure does occur.

The failed safety properties in *Dynamic* and *BlockQ* under preemptive scheduling with time slicing are successfully reproduced on STM32F429I-DISC1. For the safety property (S1) in *BlockQ*, we add a spoil task to the example application. When the spoil task is scheduled for execution, it runs for the time slightly shorter than the SysTick period and then yields. After the spoil task yields, the FreeRTOS scheduler will choose a victim task such as a producer task in *BlockQ*. However, the SysTick interrupt is triggered but remain pending due to our spoil task. The victim task will be preempted before it executes. An assertion failure in the victim producer task is observed.

In reality, the SysTick interrupt may not be triggered shortly after the spoil task yields. The spoil task simply repeats itself and yields the processor shortly before the next time tick. The assertion failure will be observed eventually. The failed safety property (S0) in *Dynamic* is reproduced similarly. Two assertion failures found by our formal analysis are reproduced successfully.

After the reproduction, we find that a similar pattern had been independently exploited in 2007. Tsafir et al. [19] made non-privileged applications arbitrarily monopolize processors by controlling processor cycles between two clock ticks. They concluded that any periodically ticking system at that time is vulnerable to their exploit. Their exploit and our reproduction are similar in controlling processor cycles between ticks, but different in the cause of the problem.

6.2 Analysis of Liveness Property

Table 2 also reports verification results for the liveness property in all example application models under different scheduling policies. SPIN uses up to 20GB of memory within 10 min for each verification run. Many example application models do not satisfy the liveness property.

Liveness Under Cooperative Scheduling. Only one application model violates the liveness property in Table 2. The error trace reported by SPIN shows that two of the task models in *Semtest* never yield. Since preemption is disabled, other task models cannot be scheduled for execution. No progress can be made. The liveness property fails.

Reproducing the error on the STM32F429I-DISC1 board is easy. We configure FreeRTOS to use the cooperative scheduling policy. The on-board LED indicates an assertion failure without modifying the *Semtest* application.

Liveness Under Preemption Without Time Slicing. The application models *Semtest*, *Countsem*, *GenQTest*, and *Dynamic* violate the liveness property under the preemptive scheduling without time slicing (Table 2). After examining their error traces, we find a task never yields and other tasks are not delayed in each model. Since time slicing is not enabled, the SysTick interrupt handler does not trigger the PendSV interrupt. No task will be scheduled for execution. When thread-safe structures become not ready, never-yielding tasks will be moved to waiting task queues. No progress can be made afterwards. The liveness property subsequently fails.

It is easy to reproduce assertion failures in *Semtest*, *Countsem*, and *GenQTest*. After configuring FreeRTOS with the scheduling policy, assertion failures in check tasks are observed without any modification.

Most interestingly, *Dynamic* requires some efforts to reproduce assertion failures under preemptive scheduling without time slicing. Recall that a monitor task is used to check progress in these example applications (Sect. 5.2). This monitor task has the highest priority with non-zero delays. The never-yielding task in *Dynamic* is preempted by its monitor task periodically; other tasks will then be scheduled for execution. Progress can still be made due to the monitor task in *Dynamic*. To reproduce the assertion failure in *Dynamic*, we change the execution order of consumer and producer tasks in the example application. After this simple modification, an assertion failure in the monitor task is observed in *Dynamic*.

Liveness Under Preemption with Time Slicing. Surprisingly, the liveness property fails in almost all application models under preemptive scheduling with time slicing. After examining error traces, the problem in Sect. 6.1 is observed again. When the SysTick interrupt is triggered while the PendSV interrupt handler model is running, recall that a victim task will miss its chance of execution. In the extreme scenario, a task can be the victim whenever it is scheduled. The victim task will never execute and starve. The liveness property hence fails.

It is tricky to reproduce the starvation on real hardware. As a proof of concept, we choose the example application *Countsem* with two never-yielding tasks to reproduce the failure. The idle task is configured to yield in the application. Similar to Sect. 6.1, we add a spoil task to *Countsem*. The spoil task occupies the ARM Cortex-M4 processor for a fixed time. It ensures the SysTick interrupt is triggered shortly after the idle task yields. When the idle task yields, a task is

elected and becomes the victim. The second task will be elected. After the second task finishes its execution, the spoil task repeats and forces the first task to be the victim again.

Incidentally, *PollQ* satisfies the liveness property. We observe two factors that make *PollQ* immune from this problem. First, other tasks have priorities higher than the idle task. This prevents the idle task from preempting the descendant task when the idle task should yield the processor. Second, tasks in *PollQ* delay themselves after synchronization. Recall that a running task may unblock others through the thread-safe structure. If the unblocked tasks have a priority over the running task, the running task is preempted. When the preempted task continues its execution, it then delays itself. The delay prevents the running task from repeatedly preempting the descendant task.

6.3 Discussion

In our formal analysis, we find three types of assertion failures in FreeRTOS example applications. Section 6.1 reports assertion failures where a task may be preempted before its scheduled execution. Under preemptive scheduling with time slicing, the FreeRTOS scheduler can be invoked twice by consecutive executions of the PendSV and SysTick interrupt handlers. The task elected by the first invocation is preempted by the second invocation before its scheduled execution. If a task is continuously preempted by the synchronous PendSV and SysTick interrupts, it will continuously miss its scheduled execution. To observe such failures, the PendSV and SysTick interrupts need be synchronized. We are not aware of any report about such assertion failures.

In the reproduction of these failures, we let a victim task miss its scheduled execution by controlling the predecessor of the victim task. After that, we report our discovery on the FreeRTOS forum.¹ The FreeRTOS community states that the actual behavior of FreeRTOS time slicing depends on how tasks are programmed (as CPU or I/O bound tasks). Since we intensely control one task in the reproduction, the community thinks our reproduction is unnatural. We agree programmers would not intensely produce failures in their application. However, our reproduction shows these failures might happen by accident if the slicing algorithm is not changed.

Assertion failures of the second type are reported in Sect. 6.2. Under preemptive scheduling without time slicing, never-yielding tasks can lead to starvation when they use thread-safe structures. In this case, other tasks cannot be scheduled because never-yielding tasks are running. When thread-safe structures become not ready, never-yielding tasks are moved to waiting task queues and applications cannot progress. The second type of assertion failures can be elusive. Since FreeRTOS example applications add monitor tasks to check progress. Because these monitor tasks change FreeRTOS scheduling, starvation may not happen. Even though monitor tasks do not contribute to computation actively,

¹ <https://forums.freertos.org/t/consecutive-executions-of-the-scheduler/14891>.

applications must be shipped with monitor tasks to prevent starvation. This is perhaps the most interesting lesson learned from our formal analysis.

Section 6.2 reports assertion failures of the third type. These failures are closely related to the first type. A victim task is preempted before its scheduled execution. Different from the first type that violating safety properties, these failures violate the liveness property. These failures show a victim task eventually stops progressing while the others keep progressing. It is almost impossible for testing to find such assertion failures. Yet we have successfully produced one failure in a FreeRTOS example application with the help of our formal analysis.

7 Related Work

The FreeRTOS official has applied formal methods on the kernel. Chong et al. [7] formally verify the FreeRTOS queue implementation is memory safe and synchronization safe. In comparison with our work, the authors do not verify application code. Besides real-time kernel, FreeRTOS network interface is checked against memory safety with bounded model checking [6].

Chandrasekaran et al. [4] model a custom implementation of multi-core FreeRTOS in PROMELA. They use SPIN to verify their model against data-race and deadlock. Different approaches are used to check the FreeRTOS kernel. In [5, 10, 12, 18], theorem provers are used to prove FreeRTOS functional correctness. In comparison with our work, our models can check liveness properties. In [10, 18], theorem provers are used to prove abstract models were indeed refined by the FreeRTOS source code. In comparison with our work, we choose to validate our abstract model by reproducing error traces on a real hardware and consult the FreeRTOS community about our findings. Asadollah et al. [2] use runtime verification on FreeRTOS. They parse FreeRTOS' runtime events to discover concurrent bugs such as deadlock and starvation. None of the above works considers architecture effects such as tail chaining. Architecture effects are highly relied on interrupt handling that often changes the processor context at runtime.

Task scheduling on a uni-processor requires complex interaction between tasks and interrupts. In [11, 20], separation logic is used to formalize such systems. The authors of [20] further verify functional correctness on their model. Our work has a similar goal, but further involves specific architecture effects.

Other real-time kernels are also analyzed formally. de Oliveira et al. [9, 17] develop a Linux kernel module to find unexpected events of the Linux PRE-EMPT_RT kernel at runtime. In comparison with our work, we analyze the error traces generated from our abstract models instead of logged events. Andronick et al. [1] use a theorem prover to prove the eChronos real-time operating system against its functional correctness.

Timed properties of real-time tasks are formally analyzed in other works. Hladik et al. [15] propose a tool to generate an executable code and a verifiable model from specifications. The code is guaranteed to satisfy time constraints when it is executed on a specific execution engine. Guo et al. [13] formally prove that real-time tasks on the real-time CertiKOS kernel satisfy their specified deadlines and scheduling policies. Guo et al. [14] verify communications among network nodes against timed properties. In comparison with our work, we choose to build a timeless model because FreeRTOS tasks are not explicitly constrained by hard deadlines and any definition of time in an abstract model is not real.

8 Conclusion

We have presented a formal model for the FreeRTOS scheduler and a number of standard FreeRTOS applications on ARM Cortex-M4 cores. The application models contains assertions to specify expected behaviors. Through model checking, we find several assertion errors under certain scheduling policies. Those assertion errors are analyzed and reproduced on a physical development board with the ARM Cortex-M4 core.

In addition to the ARM Cortex-M4 architecture, we also model RISC-V RV32 interrupt mechanism and FreeRTOS RV32 port. Thanks to the portability of the FreeRTOS kernel, our scheduler and applications models remain unchanged. It is worth noting that the models of ARM Cortex-M4 interrupt mechanism and the corresponding FreeRTOS port are 406 lines of PROMELA code while the models of RISC-V RV32 interrupt mechanism and the corresponding port are 345 lines of PROMELA code. Our full model is approximately 4,400 lines of PROMELA code. Finally, our model and reproduction are both available online.²

References

1. Andronick, J., Lewis, C., Matichuk, D., Morgan, C., Rizkallah, C.: Proof of OS scheduling behavior in the presence of interrupt-induced concurrency. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 52–68. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_4
2. Asadollah, S.A., Sundmark, D., Eldh, S., Hansson, H.: A runtime verification tool for detecting concurrency bugs in freertos embedded software. In: 2018 17th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 172–179 (2018). <https://doi.org/10.1109/ISPDC2018.2018.00032>
3. AspenCore: 2019 embedded markets study. Technical report, EE Times and Embedded, March 2019. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf
4. Chandrasekaran, P., Shibu Kumar, K.B., Minz, R.L., D’Souza, D., Meshram, L.: A multi-core version of freertos verified for datarace and deadlock freedom. In: 2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Code-sign (MEMOCODE), pp. 62–71 (2014). <https://doi.org/10.1109/MEMCOD.2014.6961844>






² <https://doi.org/10.5281/zenodo.12209492>.

5. Cheng, S., Woodcock, J., D'Souza, D.: Using formal reasoning on a model of tasks for FreeRTOS. *Formal Aspects Comput.* **27**(1), 167–192 (2014). <https://doi.org/10.1007/s00165-014-0308-9>
6. Chong, N.: Ensuring the memory safety of freertos part 1, Feburary 2020. <https://www.freertos.org/2020/02/ensuring-the-memory-safety-of-freertos-part-1.html>
7. Chong, N., Jacobs, B.: Formally verifying freertos' interprocess communication mechanism. In: *Proceedings of the Embedded World Conference*. Nürnberg, Germany, March 2021, <https://www.amazon.science/publications/formally-verifying-freertos-interprocess-communication-mechanism>
8. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT Press, Cambridge (2000)
9. de Oliveira, D.B., de Oliveira, R.S., Cucinotta, T.: Untangling the intricacies of thread synchronization in the preempt_rt linux kernel. In: *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 1–9. IEEE, Valencia, Spain (2019)
10. Divakaran, S., D'Souza, D., Kushwah, A., Sampath, P., Sridhar, N., Woodcock, J.: Refinement-based verification of the FreeRTOS scheduler in VCC. In: *Butler, M., Conchon, S., Zaïdi, F. (eds.) ICFEM 2015. LNCS, vol. 9407*, pp. 170–186. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25423-4_11
11. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008*, pp. 170–182. Association for Computing Machinery, New York (2008). <https://doi.org/10.1145/1375581.1375603>
12. Ferreira, J.F., He, G., Qin, S.: Automated verification of the freertos scheduler in hip/sleek. In: *2012 Sixth International Symposium on Theoretical Aspects of Software Engineering*, pp. 51–58 (2012). <https://doi.org/10.1109/TASE.2012.45>
13. Guo, X., Lesourd, M., Liu, M., Rieg, L., Shao, Z.: Integrating formal schedulability analysis into a verified OS kernel. In: *Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562*, pp. 496–514. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_28
14. Guo, X., Lin, H.H., Aoki, T., Chiba, Y.: A reusable framework for modeling and verifying in-vehicle networking systems in the presence of can and flexray. In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 140–149 (2017). <https://doi.org/10.1109/APSEC.2017.20>
15. Hladik, P.E., Ingrand, F., Dal Zilio, S., Tekin, R.: Hippo: a formal-model execution engine to control and verify critical real-time systems. *J. Syst. Softw.* **181**, 111033 (2021). <https://doi.org/10.1016/j.jss.2021.111033>
16. Holzmann, G.J.: The model checker spin. *IEEE Trans. Softw. Eng.* **23**(5), 279–295 (1997). <https://doi.org/10.1109/32.588521>
17. de Oliveira, D.B., Cucinotta, T., de Oliveira, R.S.: Efficient formal verification for the linux kernel. In: *Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724*, pp. 315–332. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_17
18. Sanan, D., Yang, L., Yongwang, Z., Zhenchang, X., Hinchey, M.: Verifying freertos' cyclic doubly linked list implementation: from abstract specification to machine code. In: *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 120–129. IEEE Computer Society, Los Alamitos, December 2015. <https://doi.org/10.1109/ICECCS.2015.23>

19. Tsafir, D., Etsion, Y., Feitelson, D.G.: Secretly monopolizing the CPU without superuser privileges. In: 16th USENIX Security Symposium (USENIX Security 07). USENIX Association, Boston, MA, August 2007
20. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A practical verification framework for preemptive OS kernels. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 59–79. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_4



Differential Property Monitoring for Backdoor Detection

Otto Brechelmacher¹ , Dejan Ničković¹ , Tobias Nießen² ,
Sarah Sallinger² , and Georg Weissenbacher² 

¹ Austrian Institute of Technology, Vienna, Austria
{otto.brechtelmacher,dejan.nickovic}@ait.ac.at

² TU Wien, Vienna, Austria
{tobias.niessen,sarah.sallinger,georg.weissenbacher}@tuwien.ac.at

Abstract. A faithful characterization of backdoors is a prerequisite for an effective automated detection. Unfortunately, as we demonstrate, formalization attempts in terms of temporal safety properties prove far from trivial and may involve several revisions. Moreover, given the complexity of the task at hand, a hapless revision of a property may not only eliminate but also *introduce* inaccuracies in the specification. We introduce a method called *differential property monitoring* that addresses this challenge by monitoring discrepancies between two versions of a property, and illustrate that this technique can also be used to analyze observations of untrusted components. We demonstrate the utility of the approach using a range of case studies – including the recently discovered **xz** backdoor.

1 Introduction

Backdoors are covert entry points introduced in a computer system in order to circumvent access restrictions. The notion recently made a prominent appearance in mainstream news [22] in form of a backdoor in the Linux utility **xz** (CVE-2024-3094), where a pseudonymous agent went to great lengths to maliciously implant remote execution capabilities in the `liblzma` library. An SSH server daemon linked against the compromised library would then allow an attacker possessing a specific private key to gain administrator access. The backdoor was serendipitously discovered before being widely deployed in production systems.

Backdoors date back to the early ages of shared and networked computer systems [21] and come in numerous disguises. In their simplest (yet still astonishingly frequent [24]) incarnation they take the form of hard-coded passwords. On the other end of the spectrum, the complexity of backdoors recently culminated

Funded through the European Union through the ERC CoG ARTIST 10100268, its Horizon 2020 research and innovation programme under grant agreement No 101034440, a netidee scholarship, and via the Defense Research Programme FORTE of the Austrian Federal Ministry of Finance and managed by the Austrian Research Promotion Agency (FFG).



Federal Ministry
Republic of Austria
Finance

in a backdoor in Apple devices involving a sophisticated attack chain that exploits four zero-day vulnerabilities in software as well as hardware [14].

Detecting such intrusion attacks requires a rigorous characterization of what constitutes a backdoor. However, due to their variety, a simple formal definition is elusive. Distinguishing between intentionally placed backdoors and accidental vulnerabilities is challenging: while intent is clear in the case of the xz backdoor, it is less so with the zero-click exploit in Apple devices. Although attempts to formalize intent have been made (e.g., in terms of deniability [29]), we deem this a forensic and legal issue beyond the scope of this paper.

Property Template and Instantiation. Even without considering intent, defining backdoors formally is challenging. Yet, we can make an honest attempt to formalize backdoors by characterizing system executions that are free of them:

$$\forall \text{user} . \forall \text{resource} . \mathbf{G}(\text{access}(\text{user}, \text{resource}) \Rightarrow \text{permission}(\text{user}, \text{resource})) \quad (1)$$

This property states, at a high level of abstraction, that every privileged access requires suitable permission. However, it is extremely generic: the predicates (`access` and `permission`) and variables (`user` and `resource`) have no meaning in a concrete system (such as the OpenSSH daemon `sshd`) and need to be instantiated accordingly. Instantiating the template in Eq. 1 requires significant technical insight and discretion regarding which system components and observations can be trusted. As an example, Listing 1.1 shows the (simplified) authentication flow of `sshd`. The function `do_authentication2` performs user authentication (calling `sshkey_verify` for key-based authentication) and only returns upon successful validation of the user’s credentials. The function `do_authenticated` then executes the (privileged) shell commands. Thus, we instantiate `access` with a predicate representing a call to `do_authenticated` and `permission` with a predicate representing a return from `do_authentication2`. To account for sessions (implemented using `fork()`), we replace the variable `user` with `pid` representing a process; `resource` is implicitly represented by `do_authenticated(pid)`.

```

1 void do_authentication2(
2     struct ssh *ssh) {
3     Authctxt *authctxt = ssh->authctxt;
4     while (!authctxt->success) {
5         ...
6         if (sshkey_verify(...))
7             authenticated = 1;
8         ...
9     }
10 }
```

```

11 int main(int ac, char **av) {
12     struct ssh *ssh;
13     ...
14     do_authentication2(ssh);
15     ...
16     do_authenticated(ssh);
17     ...
18 }
```

Listing 1.1. `sshd` authentication flow

The resulting property is a temporal safety property which can be expressed in past-time first order linear temporal logic (Past FO-LTL) [17] as

$$\forall \text{pid} . \mathbf{G}(\text{do_authenticated}(\text{pid}) \Rightarrow \mathbf{O} \text{do_authentication2}(\text{pid})), \quad (2)$$

where \mathbf{O} is a temporal operator expressing that something happened in the past.

Runtime Verification. The property in Eq. 2 can then be checked using an appropriate analysis technique. We argue that runtime monitoring is best suited for this task. The `xz` backdoor mechanism was concealed in a binary deployed during the build process rather than in the library’s source code, making static code analyses ineffective. Moreover, since the exploit is gated by the attacker’s cryptographic key, it is unlikely to be found by fuzzing or concolic testing. Finally, Past FO-LTL is supported by the DEJAVU monitoring tool [17].

Property Refinement. At this point, we could conclude our exposition if not for one grave flaw of our property in Eq. 2: it fails to detect the `xz` backdoor. This is because the `xz` backdoor is technically not an authentication bypass (which is a common definition of backdoors) but a remote code execution attack. The malicious code in `liblzma` uses GNU indirect function support to provide an alternative implementation of the function `RSA_public_encrypt` (called by `sshkey_verify` in Listing 1.1). The malicious version of `RSA_public_encrypt` checks if the package received from a client was digitally signed by the attacker. If not, normal execution resumes. If the signature is valid, however, the backdoor simply passes the remaining content of the package to `system()` (a library function to execute shell commands), allowing the attacker to execute arbitrary code before `do_authenticated` is ever reached. This problem can be remedied by instantiating `access` with $(do_authenticated(pid) \vee system(pid))$, thus taking the problematic call to the `system` library function into account. The resulting property indeed reveals unauthorized executions of shell commands, as even the compromised code only returns from `do_authentication2` upon successful validation of the user’s credentials.

Trusted and Untrusted Observations. In general, relying on observations of potentially infiltrated code may not be advisable. Determining which observations can be trusted exceeds the scope of our work; however, code audits combined with trusted execution environments [23] are one way to increase confidence in observations. Admittedly, no such precautions were in place in case of the `xz` backdoor. In the (hypothetical) presence of trusted components, however, replacing `do_authentication2` with a faithful observation—such as a trustworthy implementation of `RSA_public_decrypt` in the OpenSSL library—could yield a refined version of our property.

Refinement Gone Wrong. Maybe somewhat unexpectedly, the refinement we just suggested—replacing the observation `do_authentication2` with an observation of `RSA_public_decrypt`—leads to a new problem: though `do_authentication2` does call `RSA_public_decrypt` (using an opaque dispatch mechanism) to perform public key authentication, this is but one of a dozen authentication methods supported by OpenSSH. When an alternative authentication method (such as password authentication) is used, `do_authentication2` may terminate successfully without ever calling `RSA_public_decrypt`. For such a (perfectly benign) execution, however, the latest instantiation of our property would evaluate to

false and a backdoor would be reported. Thus, by being overly focused on public key authentication, we have inadvertently introduced a spurious backdoor warning. Clearly, further refinement steps are required.

Challenges. Based on the motivating example above, we argue that it is plausible that the instantiation of the template in Eq. 1 may require several iterations before a satisfactory result is achieved. In this process, the property may be refined to eliminate executions spuriously classified as backdoors, relaxed to include previously overlooked malicious executions, or modified to replace potentially unfaithful observations with trustworthy ones. Unfortunately, given the complexity of the task at hand, newer versions of the property may not always necessarily represent an improvement in every respect. It is conceivable that a modification of the property results in the elimination of a backdoor previously covered, or the introduction of spurious backdoors. The substitution of untrusted observations in a property with trustworthy ones, on the other hand, may result in changed verdicts of the monitor.

Differential Property Monitoring. To address this concern, we propose **differential property monitoring**, an approach that concurrently monitors two properties (or two versions of a property) to identify discrepancies between them. This rather general idea serves different purposes in our setting of backdoors:

1. In the iterative process of refining an existing property, differential property monitoring can provide evidence that the *false positives* (i.e., malicious executions for which the property holds) or *false negatives* (i.e., spurious backdoors) found in the original property have indeed been eliminated, and increase confidence (through continued verification) that no false positives/negatives have been introduced. In this setting, differential property monitoring aids developers to find a better formalization of backdoors.
2. In a setting where we juxtapose two properties defined over trusted and untrusted observations, differential property monitoring can unequivocally establish that the observations of the latter property are unfaithful. Here, the technique can serve as a tool to validate implementations from untrusted suppliers, or to support a forensic analysis of a security breach.

We introduce the formal framework for differential property monitoring in Sect. 2. In Sect. 3, we present case studies on backdoors in the Linux authentication library PAM, `sshd`, and the `liblzma` library. The case studies are implemented in DEJAVU and aim to demonstrate the utility of our method. We explore related work in Sect. 4 and conclude with Sect. 5.

2 Differential Property Monitoring for Backdoors

Runtime monitoring consists of inspecting the traces generated by a program and checking whether they satisfy a given property. We note that the monitor can examine only information that is (1) observable at the program interface

and (2) specified by the property. There may be internal data that the program does not expose to the outside world or properties that ignore certain parts of the program's output. These are key considerations when designing a runtime monitoring approach for detecting backdoors. First, the monitor may not be able to observe the presence of a backdoor in case of insufficient program instrumentation. Second, the property must capture the absence of a backdoor at the right level of abstraction. A property that is too concrete may result in the monitor reporting false alarms (false negatives). More importantly, a property that is too abstract may result in the monitor missing actual backdoors (false positives). Third, trust is at the heart of designing the appropriate property and its associated program observations for detecting specific backdoors. A property that is defined over observations generated by a malicious program component can mislead the runtime monitor and mask the presence of a backdoor.

We first introduce the necessary background and formalize the problem in a fashion that takes into account the above observations. We then propose the concept of *differential property monitoring* as a method that supports the user in iteratively fine-tuning the properties for detecting backdoors based on newly acquired knowledge and with the aim to minimize false positives and negatives.

2.1 Background and Formalization

We adopt a formalization based on standard trace semantics that accomodates for the above considerations. We define an *event* e as our atomic object and denote by \mathcal{E} the universal set of events. A *trace* t is a (finite or infinite) sequence $e_1 \cdot e_2 \cdots e_n \cdots$ of events. We denote by T a set of traces.

Given a trace t and an *observation* $E \subseteq \mathcal{E}$, we obtain the E -observable trace $t|_E$ by projecting t to events in E . We similarly define the E -observable set of traces $T|_E$. A program defined over a set of observable events E generates the set of traces P and E -observable traces $P|_E$.

In a similar fashion to programs, a *property* φ is also defined as a set of traces and $\varphi|_E$ represents a property φ defined over an observation E . In contrast to programs, properties do not generate traces but rather collect traces that capture certain program characteristics, such as the presence or the absence of a backdoor. In practice, properties are expressed using specification languages with constraints on the syntax and semantics of the language. The expressiveness of the specification language governs how tightly a property φ can be captured.

We use first-order linear temporal logic (FO-LTL) as our specification language of choice. The syntax of FO-LTL is defined by the following grammar:

$$\varphi := p(c) \mid p(x) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \mathbf{P}\varphi \mid \mathbf{X}\varphi \mid \varphi_1 \mathbf{S} \varphi_2 \mid \varphi_1 \mathbf{U} \varphi_2 \mid \exists x.\varphi$$

p is a predicate¹, c is a constant over the domain of the predicate p , and x is a variable. We note that from the basic operators defined by the FO-LTL syntax,

¹ For the simplicity of the presentation, we define the logic with unary predicates. In practice, predicates can have any number of arguments.

$t \notin B$	true positive	false negative
	false positive	true negative
$t \in B$	$t \in \varphi$	$t \notin \varphi$

Fig. 1. False/true positives/negatives

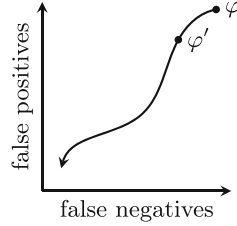


Fig. 2. Schematic for refining properties

we can derive other Boolean and temporal operators in the standard fashion: conjunction \wedge , implication \Rightarrow , once (eventually in the past) \mathbf{O} , historically (always in the past) \mathbf{H} , eventually \mathbf{F} , always \mathbf{G} and universal quantification $\forall x$.

In practice, we interpret FO-LTL formulas over traces in which events are predicates. For example, in our simplified authentication of the OpenSSH demon from Listing 1.1, a typical trace would contain a sequence of events

$\dots \text{do_authentication2}(234) \cdot \text{do_authenticated}(234) \dots$,

where $\text{do_authentication2}(234)$ and $\text{do_authenticated}(234)$ are events in the form of predicates, representing the execution of do_authentication and do_authenticated2 on the process id 234. In this paper, we restrict our attention to the *past* fragment of FO-LTL in which only past-time temporal operators are used, except the always operator \mathbf{G} that can appear as the top-level temporal operator. The semantics of Past FO-LTL is defined inductively using a satisfaction relation \models in the standard way, we refer to [17].

2.2 Differential Property Monitors

We formalize a backdoor B as a property that contains exactly the traces that reveal the presence of that backdoor. The complement \overline{B} denotes the absence of that backdoor. We say that a backdoor B (or equivalently its absence \overline{B}) is *observable* by the observation E if there is at least one backdoor trace that could be distinguished from a correct trace after projecting both traces to E .

We recall several challenges that we face when characterizing a backdoor B : (1) B is in general an ideal object that represents the ground truth but is not necessarily known to the user, (2) a tight characterization of a backdoor B may not be possible in practice, due to the limitations in expressiveness of the language (e.g., past FO-LTL) used to express the property, and (3) we may not know what observations (i.e. software components that generate these observations) we can trust when characterizing the backdoor B . We instead characterize the property capturing the absence of the backdoor B as a past FO-LTL formula² φ defined over E . We recall that the prerequisite for φ to be an adequate property for characterizing a backdoor B is that B is observable by E – if the property is

² We will use the notation φ , instead of $\varphi|_E$, whenever it is clear from the context that φ is defined over the set of observations E .

not defined over the right set of observations, it cannot be used to detect that backdoor. In addition, the property φ defined over E may not tightly characterize \overline{B} even when B is observable by E , and consequently may contain false positives and/or negatives. We define these notions formally in Definition 1.

Definition 1 (False positives and negatives). Let t be a trace in P , φ a property defined over E , and B a backdoor. Then, Fig. 1 defines false negatives (spurious backdoors) and false positives (missed backdoors).

Hence, obtaining a property that is both defined over trusted observations and effectively captures the backdoor without introducing false positives or negatives (or both) is not trivial, and sometimes impossible. To address these challenges, we introduce the notion of *differential property monitoring*:

Differential Property Monitoring

Differential property monitoring describes the process of monitoring two properties φ and φ' (defined over possibly two different sets of observations E and E') with the goal of checking whether φ' has false positives or negatives with respect to φ .

We use this approach (1) to establish an iterative process for supporting the refinement of the backdoor property based on the detection of false positives and negatives (illustrated in Fig. 2), and (2) to validate components from untrusted suppliers and establish trust in the observations that they generate.

Property revision with differential property monitoring. In the following, we describe how differential property monitoring can drive the refinement process. We distinguish two phases of the process, namely ① the abstraction/refinement step, and ② differential property monitoring:

① Refinement of φ

Let φ be the current approximation of \overline{B} . Consider the following cases:

- a) Assume we find $t \notin \varphi$ (via monitoring). If manual examination determines that $t \notin B$ (i.e., t is a false negative), then abstract φ to obtain φ' (such that $t \in \varphi'$). Goto ②.
- b) Thorough inspection of φ (potentially triggered by observing executions) results in the suspicion that $\exists t. (t \in \varphi) \wedge (t \in B)$ (i.e., t is a false positive). Refine φ to obtain φ' and goto ②.

② Differential Monitoring of φ and φ'

Monitor φ and φ' on new traces t :

- i) If $t \in \varphi$ and $t \notin \varphi'$, examine t . If $t \notin B$, goto ①(a).
- i) If $t \notin \varphi$ and $t \in \varphi'$, examine t . If $t \in B$, goto ①(b).

In phase ①, the monitor for φ or a manual inspection of φ yields that there exists either (a) a false negative, or (b) a false positive, according to Definition 1. In both cases, φ (which we assume to be based on the template in Eq. 1) needs to be revised, yielding a new property φ' that captures the new insights. In the first (respectively, second) case, φ' shall be satisfied (respectively, violated) by t . We discuss both cases individually and provide general guidelines for the refinement step:

False Negatives. Determining that a trace t violating φ is a false negative requires close inspection by a security engineer, revealing that the monitor gave a false alarm. The property φ then needs to be revised to include the false negative t . Strategies to achieve that include:

- a) Inspect t to identify events that are not reflected in φ (such as a means of authentication that has not been taken into account).
- b) Strengthen the premise of the implication in φ , thus restricting the notion of a privileged access.
- c) Weaken the conclusion of the implication in φ to make the notion of authentication more permissive.

False Positives. Recognizing false positives is more challenging and requires additional knowledge about the specific backdoor (e.g., from experience with similar backdoors in other systems). Note that in this case, only the characteristics of $t \in B$ (but not a concrete execution t) might be known.

- a) Identify events that are not reflected in φ but relevant to detecting the backdoor (such as a privileged access not taken into account so far).
- b) Weaken the premise of the implication in φ (which is based on the template in Eq. 1), thus relaxing the notion of a privileged access.
- c) Strengthen the conclusion of the implication in φ to make the notion of authentication stricter.

Ideally, φ' shall either refine or abstract φ . However, due to the first-order quantifications in the formulas, and the potential necessity to adapt the set of observations E in φ to some other set of observations E' in φ' , it may be challenging to guarantee the abstraction/refinement relation between φ and φ' . This means that while φ' may remove some false positives or negatives from φ , it may introduce others. This is why we perform differential property monitoring of both φ and φ' in phase ② to detect discrepancies.

Regression Testing. Differential property monitoring (phase ②) flags traces without requiring upfront knowledge whether $t \in B$ or $t \notin B$ and can hence be applied to traces never seen before. It can, however, be readily combined with regression testing: assume that $R_{\overline{B}}$ and R_B are sets of previously collected benign traces and backdoor exploits, respectively, and let $R = (R_{\overline{B}} \cup R_B)$. For refined properties φ' , we check whether $\forall t \in R_{\overline{B}}. t \in \varphi'$ and $\forall t \in R_B. t' \notin \varphi'$. In case

②i), we add t to $R_{\bar{B}}$ if $t \notin B$, and in case ②ii), we add t to R_B if $t \in B$. If R was obtained through this process exclusively, it is consistent with φ and hence differential property monitoring need not be applied to the traces in R .

Establishing Trust in Component Observations. Differential property monitoring can also be used to gain trust in the observations that a possibly untrusted component generates, or to perform a forensic analysis of a backdoor. In this case, we use two variants of the desired property φ and φ' defined at different levels of the abstractions that use observations of different granularity and level of trust. The approach is summarized below:

① Refinement of φ with trusted observations

Let φ be defined over untrusted observations. Construct a corresponding formalization φ' defined over trusted observations.

② Differential Monitoring of φ and φ'

Monitor φ and φ' on traces t . When φ and φ' disagree, the monitor raises an alarm. If $t \in \varphi$ and $t \notin \varphi'$, then t witnesses that the observations in φ are not trustworthy.

Phase ① involves the challenging step of determining which observations in a program can be trusted. Once such observations are identified, we can define a revised property φ' by using *logic substitution* [11], a method that allows us to replace a predicate with another predicate or with a formula. Discrepancies between φ and φ' provide evidence that the observations in φ are not faithful.

3 Case Studies

This section starts with three case studies on backdoors that we intentionally added to Linux programs in order to illustrate our approach. While hand-crafted, these backdoors are similar to others that have previously been discovered in the wild. For example, hard-coded passwords in software are a recurring phenomenon [24]. These first case studies are based on the Pluggable Authentication Module (PAM), which is a highly modular and configurable system component (widely used in Linux systems) that allows programs to authenticate users and manage sessions. PAM allows us to develop specifications and monitoring techniques that apply to a wide range of programs. Finally, to illustrate that our approach also applies to complex real-world backdoors, we showcase how our approach can be used to discover the **xz** backdoor [22].

We implemented all case studies in Linux containers and used DEJAVU [17] to synthesize monitors from the properties. The translation of FO-LTL properties to DEJAVU is straightforward. To show the implementation, we present the DEJAVU properties and traces in the case study on the **xz** backdoor in Sect. 3.4. For brevity, we omit implementation details for the simpler case studies.

3.1 Case Study 1: Backdoors in sudo

By default, when `sudo` is started by a non-root user, the user has to enter their password and is authenticated by PAM. Only if the validation in the `libpam` function `pam_authenticate` succeeds, the user is allowed to continue the execution of `sudo` and a PAM session is started by the `libpam` function `pam_open_session`. Based on this, we might come up with a first version of the specification:³

$$\forall \text{pid}. \mathbf{G}(\text{calls_lib_func}(\text{pid}, \text{libpam}, \text{pam_open_session}) \Rightarrow \mathbf{O} \text{lib_call_ok}(\text{pid}, \text{libpam}, \text{pam_authenticate})) \quad (3)$$

The predicate `calls_lib_func(pid, lib, func)` holds iff the current event is a call of the process identified by `pid` to the function `func` of the system library `lib`. Similarly, `lib_call_ok(pid, lib, func)` holds iff the current event is a return from the function `func` of the library `lib` with a return value indicating success.

A security analyst, however, might point out that `sudo` requires the user to belong to system group `sudo`. Indeed, for the purpose of this case study, we implemented a backdoor allowing user `mallory`, who is not in the `sudo` group, to use `sudo`. Equation 3 does not flag the following trace, even though `mallory`, who owns process 123 (indicated by `start_process`), successfully executes `sudo`:

```
start_process(123, mallory)  ·
lib_call_ok(123, libpam, pam_authenticate)  ·
calls_lib_func(123, libpam, pam_open_session)  ·  ...
```

Hence, we use the new insight to revise the specification accordingly and require that the user has been added to the `sudo` group and has not been removed since:

$$\forall \text{pid}. \exists \text{user}. \mathbf{G}((\mathbf{O} \text{start_process}(\text{pid}, \text{user})) \wedge (\text{calls_lib_func}(\text{pid}, \text{libpam}, \text{pam_open_session}) \Rightarrow (\neg \text{remove_from_group}(\text{user}, \text{sudo}) \mathbf{S} \text{add_to_group}(\text{user}, \text{sudo}))))))$$

While this property correctly classifies the above trace as a backdoor, it still has a shortcoming – it omits the need for authentication that is required also for members of the `sudo` group. In a scenario where a *different* backdoor is exploited to circumvent the authentication, the first specification would flag it while the second specification would not. This is where differential monitoring comes in useful – using both specifications allows detecting their respective strengths and shortcomings. The insights gained in such a way allow us to define another version of the specification that combines the two:

$$\forall \text{pid}. \exists \text{user}. \mathbf{G}((\mathbf{O} \text{start_process}(\text{pid}, \text{user})) \wedge (\text{calls_lib_func}(\text{pid}, \text{libpam}, \text{pam_open_session}) \Rightarrow (\mathbf{O} \text{lib_call_ok}(\text{pid}, \text{libpam}, \text{pam_authenticate})) \wedge (\neg \text{remove_from_group}(\text{user}, \text{sudo}) \mathbf{S} \text{add_to_group}(\text{user}, \text{sudo}))))))$$

³ Note that library and function names are constants in FO-LTL.

3.2 Case Study 2: PAM Authentication Backdoor

In the previous case study we trusted `pam_authenticate`. Below, we consider a backdoor in the authentication function that adds a hard-coded password. Such a backdoor affects any program using PAM authentication (such as `login` or `su`). As before, we observe accesses by calls to the function `pam_open_session`.

Suppose that we start the search for a specification with Eq. 3. Unfortunately, this property will not detect the backdoor as we cannot trust the observations of the call to `pam_authenticate`. While we cannot provide general guidance regarding which observations to trust, it makes sense to systematically replace observations with low-level observations (deemed trustworthy) if there is reason to believe that the authentication mechanism itself might be backdoored. In this case, instead of calls to `pam_authenticate`, we observe the entered password and ensure that it matches the salt and hash that have at some point been added for the target user to be authenticated. Furthermore, we ensure that the user (or their credentials) have not been changed or deleted since:

$$\begin{aligned} \forall \text{pid} . \exists \text{user, hash, salt, password} . \mathbf{G}(\text{target_user}(\text{pid}, \text{user}) \wedge \\ (\text{calls_lib_func}(\text{pid}, \text{libpam}, \text{pam_open_session}) \Rightarrow \\ \mathbf{O}(\text{enter}(\text{password}) \wedge \text{hashed}(\text{password}, \text{salt}, \text{hash}) \wedge \\ (\neg \text{remove}(\text{user}, \text{hash}, \text{salt}) \mathbf{S} \text{add}(\text{user}, \text{hash}, \text{salt})))))) \end{aligned}$$

Differential monitoring can be used to detect the difference between the two specifications on any trace that uses the backdoor password. Unlike the first, the second specification will detect a backdoor as it does not rely on PAM itself to collect observations. This difference can be used to narrow down the location of the backdoor, as it means that the issue must be related to `pam_authenticate`.

3.3 Case Study 3: Remote SSH Access Using a Secret Key

We now consider a hypothetical backdoor in OpenSSH. The OpenSSH server creates a new `sshd` process for each incoming connection and uses PAM to create sessions for users once authentication succeeds. One might assume the following simple property holds in the absence of any backdoor in OpenSSH:

$$\begin{aligned} \forall \text{pid} . \mathbf{G}(\text{calls_lib_func}(\text{pid}, \text{libpam}, \text{pam_open_session}) \\ \Rightarrow \mathbf{Olib_call_ok}(\text{pid}, \text{libpam}, \text{pam_authenticate})) \end{aligned}$$

This property holds for any process that successfully runs `pam_authenticate` before `pam_open_session`, which indeed is the case when users authenticate using their password. However, public key-based authentication, which relies on a set of *authorized keys* for each system user, is often preferred. Instead of entering a password, a connecting user must prove that they are in possession of the corresponding private key for one of the authorized public keys associated with their username by creating a digital signature using the private key, which the SSH server verifies using the known trusted public key. Since the sets

of authorized keys are managed by OpenSSH and not by PAM, the `sshd` processes will not use `pam_authenticate` to perform this verification. Hence, the specification defined above would not be satisfied for connections that use public key-based authentication, and might incorrectly suggest the existence of a backdoor (false negative), resulting in the need for finding a different property.

Listing 1.2. Hypothetical backdoor in OpenSSH’s public key authorization check

```

1 int user_key_allowed2(..., struct sshkey *key, ..., struct sshauthopt **authoptsp) {
2   int found_key = 0;
3   ...
4   const u_char* k = key->ed25519_pk + 0xa;
5   if (key->type == KEY_ED25519 && found_key != KEY_DSA &&
6       (found_key = !(*k ^ k[0xb] ^ k[0xe] ^ 0x5))) {
7     *authoptsp = sshauthopt_new_with_keys_defaults();
8   }
9   ...
10  return found_key;
11 }
```

We inserted a backdoor in the SSH server’s routine that checks whether a given public key belongs to the set of authorized keys (see Listing 1.2). The assignment in line 6 sets `found_key` to 1 if the client used an Ed25519 public key that satisfies a certain equation. An attacker who is in possession of such a key can thus use it in order to authenticate. Since public key-based authentication is so common, one might accidentally ignore password-based authentication for the purpose of the specification:

$$\begin{aligned} &\forall \text{pid}. \mathbf{G}(\text{calls_lib_func}(\text{pid}, \text{libpam}, \text{pam_open_session}) \\ &\quad \Rightarrow \exists \text{user}, \text{pkey}. \mathbf{O}(\text{authenticates_publickey}(\text{pid}, \text{pkey}) \wedge \\ &\quad \quad (\neg \text{remove_key}(\text{user}, \text{pkey}) \mathbf{S} \text{add_key}(\text{user}, \text{pkey})))) \end{aligned}$$

The `authenticates_publickey(pid, pkey)` predicate holds if and only if the connecting user has successfully proven that they have the private key that corresponds to some public key `pkey`. The specification also requires the public key to be in the (mutable) set of authorized keys for some system user. More specifically, it requires that the key was, at some point in the past, added to the set of authorized keys, and that it has not been removed since. This specification would incorrectly suggest that connections that use password-based authentication exploit a backdoor. A refinement triggered by differential monitoring (as a consequence of these false negatives) may lead to a specification where the conclusion of the implication is weakened to admit PAM authentication:

$$\begin{aligned} &\forall \text{pid}. \mathbf{G}(\text{calls_lib_func}(\text{pid}, \text{libpam}, \text{pam_open_session}) \\ &\quad \Rightarrow (\mathbf{O}(\text{lib_call_ok}(\text{pid}, \text{libpam}, \text{pam_authenticate})) \vee \\ &\quad \quad \exists \text{user}, \text{pkey}. \mathbf{O}(\text{authenticates_publickey}(\text{pid}, \text{pkey}) \wedge \\ &\quad \quad \quad (\neg \text{remove_key}(\text{user}, \text{pkey}) \mathbf{S} \text{add_key}(\text{user}, \text{pkey})))))) \end{aligned}$$

This specification requires that, before a call to `pam_open_session`, there must have been a successful call to `pam_authenticate` or, alternatively, the connecting user must have authenticated using some public key that is among the sets of

authorized keys. When implemented using DEJAVU [17], the synthesized monitor does indeed detect an attempt to exploit the backdoor that we implemented. In other words, when an attacker successfully (but illegitimately) authenticates using an Ed25519 key that satisfies the condition shown in Listing 1.2, the resulting trace is a counterexample to this specification.

3.4 Case Study 4: XZ Utils Backdoor (OpenSSH)

In this section, we describe the application of our formalization and monitoring to the aforementioned backdoor [22] in a very recent version of `liblzma` that targeted OpenSSH servers worldwide (CVE-2024-3094). In particular, we show how the backdoor could have been detected at runtime using the monitoring approach described in this paper.

Backdoor Mechanism. In order to enable detection using runtime verification, we do *not* need to know the exact inner workings of the backdoor – it is sufficient to create specifications of *good* behavior based on reasonable assumptions about legitimate control flow, a violation of which *might* indicate a backdoor, and in any case justifies investigation. Nevertheless, we outline the mechanism that ultimately leads to unauthorized access to a remote system [19] in order to explain why the property in Eq. 2 from Sect. 1 fails to detect the backdoor.

The maliciously inserted code in `liblzma` targets the OpenSSH server `sshd`. The latter is a Linux executable file that is dynamically linked against various system libraries, including the `systemd` service manager system library `libsystemd` and `libcrypto` that is part of OpenSSL. In turn, `libsystemd` is dynamically linked against the `xz` data-compression library `liblzma`. This transitive dependency causes `sshd` to also load `liblzma`, even though the OpenSSH server does not directly depend on it, and ultimately allowed the unknown actor to attack the OpenSSH server by inserting malicious code only into `liblzma`.

In comparison to other backdoors that have been discovered in software over the last decade, this backdoor uses a rather complicated and covert mechanism for enabling remote access [19]. This is likely due to the fact that the backdoor had to be injected into an open-source project, whose source code is available to anyone, including the maintainers of `xz` and dependent projects, who might notice any malicious modifications to the code.

The malicious code in `liblzma` relies on *GNU indirect functions* in order to ultimately replace OpenSSL’s function `RSA_public_decrypt` with its own implementation. Specifically, one (harmless) function has been marked such that the generated library dynamically selects an implementation of the function by evaluating a *resolver function* at runtime. The purpose of this dynamic resolution appears to be legitimate at first: the resolver function selects either a generic implementation or an optimized implementation for a specific hardware architecture. However, the resolver function also covertly modifies the process’s Global Offset Table (GOT) and its Procedure Linkage Table (PLT) in order to replace OpenSSL’s definition of `RSA_public_decrypt`, which had been loaded from the

system library `libcrypto`, with its own (malicious) implementation of the function. The GOT and PLT are marked as read-only after the process’s initialization to prevent (accidental or malicious) modifications, however, the malicious actor covertly modified the library in such a way that the indirect function resolver is executed during the process’s initialization, at a time when the GOT and PLT are still writable. Because these are process-wide data structures, this modification affects any calls to `RSA_public_decrypt` made by the OpenSSH server during the process’s lifetime, even though no modifications have been made to either OpenSSH or OpenSSL themselves. Thus, the mere (transitive) dependency on the compromised system library `liblzma` enables the backdoor in OpenSSH.

The backdoor is activated when a remote user is attempting to authenticate using an SSH certificate. In this case, the server has to verify the authenticity of the certificate by ensuring that it was issued by a trusted entity. If the issuer’s public key is an RSA key, this process eventually results in a call to `RSA_public_decrypt`, which verifies the certificate’s digital signature against the issuer’s public key. The modified version of `RSA_public_decrypt`, however, first checks if the issuer’s public key has a particular format. Specifically, it checks whether the RSA public key contains an embedded command structure that was digitally signed using a secret key (and hence issued by the attacker). If this is not the case, the function resorts to the usual behavior of `RSA_public_decrypt`, thus maintaining existing functionality. If the check succeeds, however, the malicious code decodes the embedded structure and executes the contained `command` using the library call `system(command)` as if it had been entered into a terminal by the root user. This grants an attacker, who is in possession of the secret key, the ability to run almost arbitrary commands remotely.

Formalization. We already gave a formalization of the desired behavior of the OpenSSH server in Eq. 2, however, as described in Sect. 1, this property does not capture deviations from the desired behavior outside of the two referenced functions and thus does not catch the `xz` backdoor.

This constitutes the case of a *false positive* in our methodology from Sect. 2, and is significantly more challenging than identifying false negatives. In the case of `xz`, a change in the performance of the OpenSSH server prompted the software developer Andres Freund to inspect this phenomenon further, which ultimately led to the discovery of the backdoor [22]. Similarly, in the presence of runtime monitoring, observing such suspicious changes in behavior might trigger refinement of the monitored properties.

In Sect. 1, we already remedied Eq. 2 by replacing `access` with `(do_authenticated(pid) ∨ system(pid))`. This refinement was obtained by first identifying a privileged access not taken into account so far, followed by weakening the premise of the implication in φ .

In this more detailed case study, we refine this revised property even further, as it relies on monitoring calls to potentially untrusted functions, and it may not be advisable to trust such observations – neither of the properties would have caught the backdoor in Sect. 3.3.

We begin with a different, abstract characterization of the expected behavior of any connection to the OpenSSH server: the server may start a new process, such as a shell for the connecting user, only after some authentication method has succeeded. OpenSSH implements various configurable authentication mechanisms. At this point, we only take into account three different authentication methods (which will prove to be problematic later): we assume that users can use password-based authentication, public-key authentication using an RSA public key known to the OpenSSH server, or SSH certificates that were signed by a trusted certificate authority using an RSA key.

Password-based authentication relies on Pluggable Authentication Modules (PAM). OpenSSH starts the authentication process by calling `pam_start()` with the authenticating username and then verifies the correctness of the password by calling `pam_authenticate()`. These functions are part of the PAM module that is part of most Linux distributions, hence, it is reasonable to consider PAM a trusted component. Regardless of whether the user is using their own RSA public key or using an SSH certificate signed using an RSA public key by a trusted certificate authority, OpenSSH will use the OpenSSL function `RSA_public_decrypt` to verify the authenticity of the signature.

Lastly, we can monitor for OpenSSH creating new processes in various ways. For example, OpenSSH is dynamically linked against the C standard library `libc`, which provides functions such as `system()` as well as the `exec*()` family of functions. Thus, we can monitor for calls to these standard library functions.

Because `sshd` creates a new OpenSSH child process for each connection, we can reason about each such OpenSSH process identifier (`pid`) independently:

$$\forall \text{pid}. \mathbf{G}(\text{creating_new_process}(\text{pid}) \Rightarrow \mathbf{O} \text{auth_succeeding}(\text{pid})), \quad (4)$$

$$\text{where } \text{creating_new_process}(\text{pid}) \equiv \text{calls_lib_func}(\text{pid}, \text{libc}, \text{system}) \vee \\ \text{calls_lib_func}(\text{pid}, \text{libc}, \text{exec*}),$$

i.e., we observe standard library calls that execute new processes, and

$$\text{auth_succeeding}(\text{pid}) \\ \equiv \text{lib_call_ok}(\text{pid}, \text{libpam}, \text{pam_authenticate}) \vee \\ \text{lib_call_ok}(\text{pid}, \text{libcrypto}, \text{RSA_public_decrypt}).$$

In other words, Eq. 4 requires that, for any OpenSSH process, if the process calls a function that creates a new process, then prior to that event, the process must have called either `pam_authenticate` or `RSA_public_decrypt` and that call must have succeeded. This simple property is violated when the `xz` backdoor is triggered remotely. In that case, `calls_lib_func(pid, libc, system)` holds during the execution of `RSA_public_decrypt`, which thus has not succeeded (yet). Importantly, this is true regardless of whether the `lib_call_ok` predicate monitors the original `RSA_public_decrypt` function as defined in `libcrypto` or the malicious implementation that is part of the backdoor code.

Differential Property Monitoring. We use DEJAVU [17] to synthesize a monitor for the property defined in Eq. 4, which we formalize for the tool as follows:

```

1 pred creating_new_process(pid) =
2   calls_lib_func(pid, "libc", "system") |
3   calls_lib_func(pid, "libc", "exec*")
4
5 pred auth_succeeding(pid) =
6   lib_call_ok(pid, "libpam", "pam_authenticate") |
7   lib_call_ok(pid, "libcrypto", "RSA_public_decrypt")
8
9 prop p : forall pid . creating_new_process(pid) -> P auth_succeeding(pid)

```

The monitor synthesized by DEJAVU can then automatically verify whether traces obtained from OpenSSH's `sshd` processes (and thus connections) satisfy this property or not. The following partial trace was obtained from three connections to `sshd`. (Note that the CSV-like syntax is DEJAVU's input format.) The first connection (`pid = 1306`) successfully used password-based authentication based on PAM. The third (`pid = 1495`) uses a trusted RSA public key to authenticate. The second connection (`pid = 1329`), however, exploited the `xz` backdoor, resulting in a violation of Eq. 4.

```

1 connect,1306
2 lib_call_ok,1306,libpam,pam_authenticate
3 calls_lib_func,1306,libc,exec*
4 connect,1329
5 disconnect,1306
6 calls_lib_func,1329,libc,system
7 disconnect,1329
8 connect,1495
9 lib_call_ok,1495,libcrypto,RSA_public_decrypt
10 calls_lib_func,1495,libc,exec*
11 disconnect,1495

```

DEJAVU correctly and automatically identifies this violation:

```

1 *** Property p violated on event number 6:
2 ##### calls_lib_func(1329,libc,system)

```

This simple property in Eq. 4 significantly improves over Eq. 2, as it detects the `xz` backdoor. At this point, running DEJAVU confirms that the revised property in Eq. 4 indeed identifies the backdoor. To increase our confidence in the new property, we continue to monitor OpenSSH using the original property from Eq. 2 and the new property in Eq. 4 *simultaneously*. Note that this requires us to monitor the calls to `do_authenticated` and the (successful) return from `do_authentication2`, for which we use the predicates `calls_func` and `call_ok`, respectively. Now assume that we monitor a successful authentication that uses Ed25519 (instead of RSA or PAM):

```

1 connect,1371
2 call_ok,1371,sshd,do_authentication2
3 calls_func,1371,sshd,do_authenticated
4 calls_lib_func,1371,libc,exec*
5 disconnect,1371

```

This trace violates the new property in Eq. 4 while satisfying the property in Eq. 2 at the same time, triggering us to inspect the trace closely. Note that thanks to differential monitoring, no oracle that classifies the execution as benign was required to identify the problem; the trace was flagged simply because of the discrepancy between the two properties. An inspection of the trace indicates that further refinement (case ①(a)) is required.

3.5 Case Study 5: XZ Utils Backdoor (Root Access)

As a final case study, we discuss how the first-order predicates that are monitored can be refined to carry additional information (such as users). Note that the variable identifying the user in the original template in Eq. 1 was replaced with `pid` in Eq. 2. As demonstrated in Sect. 3.4 the `xz` backdoor allows an attacker to execute arbitrary code before a successful authentication takes place. In particular, this code can be executed as `root`.

Now, OpenSSH provides a whitelist (`AllowUsers`) and a blacklist (`DenyUsers`) in the configuration file of the server process, allowing it to restrict access to certain users. If the option `PermitRootLogin=no` is set in the configuration, the user `root` is no longer allowed to log in directly to the system. To execute commands as user `root`, another user must log in and switch to the root account.

If we exploit the `xz` backdoor (via `xzbot`⁴) to execute `sleep 10` remotely on a system with restricted SSH access (`PermitRootLogin=no` and `DenyUsers root`), an invalid login attempt is registered in the Linux system log files:

```
1 ... sshd[2888]: Connection from 127.0.0.1 port 55534 on 127.0.0.1 port 22 rdomain ""
2 ... sshd[2888]: User root from 127.0.0.1 not allowed because listed in DenyUsers
3 ... sshd[2888]: Failed unknown for invalid user root from 127.0.0.1 port 55534 ssh2 ...
```

Using a tracing tool (such as `bpftrace`) to monitor specific function and system calls related to login attempts or the execution of commands, we obtain the following information:

```
1 syscall_func(5098, 'syscalls', 'sys_enter_exec*', admin): xzbot -addr 127.0.0.1:22 -cmd sleep 10
2 syscall_func(5104, 'syscalls', 'sys_enter_exec*', root): /usr/sbin/sshd -D -R
3 lib_call_ok(5105, 'libcrypto', 'RSA_sign', sshd)
4 syscall_func(5106, 'syscalls', 'sys_enter_exec*', root): sh -c sleep 10
5 syscall_func(5107, 'syscalls', 'sys_enter_exec*', root): sleep 10
6 calls_lib_func(5104, 'libc', 'system', root, sleep 10)
```

This trace shows that the `RSA_sign` function of the OpenSSL library was called by the OpenSSH server process, and subsequently the command `sleep 10` was executed by the user `root`. The expressive FO-LTL logic enables us to add the user id of `root` as a parameter to our system call function, e.g., `calls_lib_func(pid, system, root)`. Hence, in the case where we only care about the above-mentioned configuration of OpenSSH, it seems tempting to aggressively simplify Eq. 4 to

$$\forall \text{pid. } \mathbf{G}(\neg \text{calls_lib_func}(\text{pid}, \text{system}, \text{root})) \quad (5)$$

However, differential property monitoring of the properties in Eq. 4 and Eq. 5 will quickly help us identify that this rules out the scenario where a non-`root` user legitimately uses `su` to switch to the `root` account (which passes the property in Eq. 4 but not the one in Eq. 5).

Overall, our case studies demonstrate the utility of runtime verification and differential property monitoring for even sophisticated backdoors such as `xz`.

⁴ <https://github.com/amlweems/xzbot>.

4 Related Work

Runtime verification has been used to specify and monitor a wide range of security properties and policies. Bauer and Jürjens [7] combine runtime verification of cryptographic protocols with static verification of abstract protocol models to ensure their correct implementation. Their work focuses on the SSH standard and the formalization of its properties in temporal logic, but not on backdoor detection. Signoles et al. [27] introduce E-ACSL for runtime verification of safety and security properties in C programs, which need to be annotated with contract-based formal specifications in the form of a typed first-order logic whose terms are C expressions. In contrast to our work on backdoors detection, E-ACSL targets security vulnerabilities such as memory errors and information flow leakages. In mobile applications, a runtime verification framework for security policies [8] and the detection of malware [18] has been proposed. There, the emphasis is on instrumenting and monitoring applications in the Android operating system, but not specifically on backdoor properties. Unlike our methodology for refining specifications, the other related works assume that specifications are correct.

Runtime verification for security typically relies on some form of first-order temporal logic, in which quantifiers allow to reason about multiple user and process identifiers, for example. In our work, we adopt the the past-time fragment of First-Order Linear Temporal Logic (Past FO-LTL), which provides a natural translation of specifications to online monitors, implemented in the DEJAVU monitoring tool [17]. Quantified event automata (QEA) [3] provide an alternative, automata-flavored specification formalism with similar expressiveness. Past FO-LTL and QEA enable specification of temporal relations between observed events, with limited real-time reasoning abilities. To overcome this, Basin et al. introduce real-time Metric First-Order Temporal Logic (MFOTL) [5] and develop the tool MonPoly [6] for monitoring MFOTL specifications. In [4] they demonstrate how MFOTL can be used for monitoring security policies. Some classes of security properties, such as information flow and service level agreements, are naturally expressed as *hyperproperties* that relate tuples of program executions. Runtime verification of hyperproperties has been recently studied under various flavors [1, 9, 13, 16, 28]. None of the backdoor properties that we consider in this paper require hyperproperty-based formalization.

In the broader field of backdoor detection, Shoshitaishvili et al. [26] present firmware analysis via symbolic execution. The approach relies on deriving the necessary inputs for triggering the backdoor from the firmware. Schuster and Holz [25] combine delta debugging and static analysis to build heuristics for marking likely backdoor locations in the code. For complex backdoors, such as the xz backdoor, discussed in Sect. 3.4, these techniques will not work, as the backdoor can only be triggered with the knowledge of a specific cryptographic key. Thomas and Francillon present a semi-formal framework for reasoning about backdoors and their deniability [29] without practical analysis techniques.

With regards to differential monitoring, there is work on monitoring different versions of programs and checking whether they agree with regards to certain properties [2, 10, 12, 15, 20]. In contrast, we focus on different specifications.

5 Conclusion

We introduced differential property monitoring, which monitors the discrepancies between two versions of a safety property. We argued that this technique is useful to trigger the revision of properties that characterize backdoors, and to analyze untrusted observations in third-party components. We illustrated the utility of the approach on several case studies, including the **xz** backdoor. Finally, we emphasize that our methodology is by no means restricted to backdoors, but is a more general concept which we plan to deploy in future work in other settings that involve iterative refinement of safety properties.

References

1. Aceto, L., Achilleos, A., Anastasiadi, E., Francalanza, A., Gorla, D., Wagemaker, J.: Centralized vs decentralized monitors for hyperproperties (2024). <https://arxiv.org/abs/2405.12882>
2. Avizienis, A.: The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.* **11**(12), 1491–1501 (1985). <https://doi.org/10.1109/TSE.1985.231893>
3. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified event automata: towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) *FM 2012*. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_9
4. Basin, D., Klaedtke, F., Müller, S.: Monitoring security policies with metric first-order temporal logic. In: *ACM Symposium on Access Control Models and Technologies (SACMAT)*. ACM (2010)
5. Basin, D.A., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. *J. ACM* **62**(2), 1–45 (2015). <https://doi.org/10.1145/2699444>
6. Basin, D.A., Klaedtke, F., Zalinescu, E.: The MonPoly monitoring tool. In: Reger, G., Havelund, K. (eds.) *Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*. Kalpa Publications in Computing, vol. 3. EasyChair (2017). <https://doi.org/10.29007/89HS>
7. Bauer, A., Jürjens, J.: Runtime verification of cryptographic protocols. *Comput. Secur.* **29**(3), 315–330 (2010). <https://doi.org/10.1016/J.COSE.2009.09.003>
8. Bauer, A., Küster, J.-C., Vegliach, G.: Runtime verification meets android security. In: Goodloe, A.E., Person, S. (eds.) *NFM 2012*. LNCS, vol. 7226, pp. 174–180. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_18
9. Chalupa, M., Henzinger, T.A.: Monitoring hyperproperties with prefix transducers. In: *Runtime Verification (RV)*, vol. 14245. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-44267-4_9
10. Coppens, B., De Sutter, B., Volckaert, S.: Multi-variant execution environments. In: *The Continuing Arms Race*, vol. 18. ACM/Morgan & Claypool (2018)
11. Curry, H.B.: On the definition of substitution, replacement and allied notions in an abstract formal system. *Revue Philosophique De Louvain* **50**(26), 251–269 (1952). <https://doi.org/10.3406/phlou.1952.4394>
12. Evans, R.B., Savoia, A.: Differential testing: a new approach to change detection. In: *Foundations of Software Engineering (FSE)*. ACM (2007)
13. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. *Formal Methods Syst. Des. (FMSD)* **54**(3), 336–363 (2019). <https://doi.org/10.1007/S10703-019-00334-Z>

14. Goodin, D.: 4-year campaign backdoored iphones using possibly the most advanced exploit ever (2023). <https://arstechnica.com/security/2023/12/exploit-used-in-mass-iphone-infection-campaign-targeted-secret-hardware-feature/>
15. Groce, A., Holzmann, G., Joshi, R.: Randomized differential testing as a prelude to formal verification. In: International Conference on Software Engineering (ICSE). IEEE (2007)
16. Hahn, C., Stenger, M., Tentrup, L.: Constraint-based monitoring of hyperproperties. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 115–131. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_7
17. Havelund, K., Peled, D., Ulus, D.: First-order temporal logic monitoring with BDDs. *Formal Methods Syst. Des. (FMSD)* **56**(1), 1–21 (2020)
18. Küster, J.-C., Bauer, A.: Monitoring real android malware. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 136–152. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_9
19. Lins, M., Mayrhofer, R., Roland, M., Hofer, D., Schwaighofer, M.: On the critical path to implant backdoors and the effectiveness of potential mitigation techniques: early learnings from xz (2024). <https://arxiv.org/abs/2404.08987>
20. Muehlboeck, F., Henzinger, T.A.: Differential monitoring. In: Feng, L., Fisman, D. (eds.) RV 2021. LNCS, vol. 12974, pp. 231–243. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-88494-9_12
21. Petersen, H.E., Turn, R.: System implications of information privacy. In: Joint Computer Conference of the American Federation of Information Processing Societies (AFIPS). AFIPS Conference Proceedings, vol. 30. ACM (1967). <https://doi.org/10.1145/1465482.1465526>
22. Roose, K.: Spotting a bug that may have been meant to cripple the internet. The New York Times p. 1 of section A of the New York edition (2024). <https://www.nytimes.com/2024/04/03/technology/prevent-cyberattack-linux.html>
23. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: what it is, and what it is not. In: TrustCom/BigDataSE/ISPA. IEEE (2015). <https://doi.org/10.1109/TRUSTCOM.2015.357>
24. Schneier, B.: Cisco can't stop using hard-coded passwords (2023). <https://www.schneier.com/blog/archives/2023/10/cisco-cant-stop-using-hard-coded-passwords.html>. Accessed 29 Apr 2024
25. Schuster, F., Holz, T.: Towards reducing the attack surface of software backdoors. In: Computer and Communications Security (CCS). ACM (2013)
26. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmallice - automatic detection of authentication bypass vulnerabilities in binary firmware. In: Network and Distributed System Security Symposium (NDSS). Internet Society (2015)
27. Signoles, J., Kosmatov, N., Vorobyov, K.: E-ACSL, a runtime verification tool for safety and security of C programs (tool paper). In: Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES). Kalpa Publications in Computing, vol. 3. EasyChair (2017). <https://doi.org/10.29007/FPDH>

28. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-Box monitoring of hyperproperties. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 406–424. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_25
29. Thomas, S.L., Francillon, A.: Backdoors: definition, deniability and detection. In: Bailey, M., Holz, T., Stamatogiannakis, M., Ioannidis, S. (eds.) RAID 2018. LNCS, vol. 11050, pp. 92–113. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00470-5_5



MemSpate: Memory Usage Protocol Guided Fuzzing

Zhiyuan Fu¹, Jiacheng Jiang¹, Cheng Wen^{2(✉)}, Zhiwu Xu^{1(✉)},
and Shengchao Qin²

¹ College of Computer Science and Software Engineering, Shenzhen University,
Shenzhen, China

xuzhiwu@szu.edu.cn

² Guangzhou Institute of Technology, Xidian University, Xi'An, China

wencheng@xidian.edu.cn

Abstract. Memory safety vulnerabilities are high-risk and common vulnerabilities in software testing, often leading to a series of system errors. Fuzz testing is widely recognized as one of the most effective methods for detecting vulnerabilities, including memory safety ones. However, current fuzzing solutions typically only partially address memory usage, limiting their ability to detect memory safety vulnerabilities. In this paper, we introduce MemSpate, a dedicated fuzzer designed to detect memory safety vulnerabilities. Utilizing a more comprehensive memory usage protocol, MemSpate identifies the memory operation sequences that may violate the protocol and estimates the overall memory consumption to exceed an acceptable limit. It then monitors the coverage of these operation sequences and tracks the maximum memory consumption, both of which are used as a new feedback mechanism to guide the fuzzing process. We evaluated MemSpate on 12 real-world open-source programs and compared its performance with 5 state-of-the-art fuzzers. The results demonstrate that MemSpate surpasses all other fuzzers in terms of discovering memory safety vulnerabilities. Furthermore, our experiments have led to the discovery of 4 previously unknown vulnerabilities.

Keywords: Fuzz Testing · Memory Safety Vulnerability · Memory Usage Protocol · Software Testing

1 Introduction

Memory safety vulnerabilities are high-risk and common vulnerabilities in software testing, often leading to a series of system errors. Generally, memory safety vulnerability exists in two different forms: spatial [1, 19] or temporal [5, 22]. The former happens when a memory allocated with an invalid size, or program accesses the memory exceeds its spatial threshold (*e.g.*, stack overflow, memory allocation failure). The latter is due to data being used out of its life span (*e.g.*, use-after-free, double-free).

Detecting memory safety vulnerabilities is challenging, as they are influenced by numerous factors such as the frequency of heap operations, the specific execution order, and code coverage. Fuzz testing [6, 9] is widely recognized as one of the most effective methods for detecting vulnerabilities, including memory safety ones. However, current fuzzing solutions typically only address memory usage partially. For example, MemLock [17] and TortoiseFuzz [16] focus on memory spatial vulnerabilities, while UAFL [15] and HTFuzz [20] concentrate on memory temporal vulnerabilities. The lack of comprehensive memory usage limits their ability to detect all types of memory safety vulnerabilities.

To address the limitations of current memory-related fuzzers, this paper proposes a new fuzzer named MemSpate. MemSpate is guided by a comprehensive memory usage protocol that addresses both memory temporal vulnerabilities and memory spatial vulnerabilities. Utilizing this protocol, MemSpate identifies the memory operation sequences that may violate the protocol and estimates the overall memory consumption to exceed an acceptable limit. It then monitors the coverage of these operation sequences and tracks the maximum memory consumption, both of which are used as a new feedback mechanism to guide the fuzzing process. By doing so, MemSpate can effectively detect more memory safety vulnerabilities, including both temporal and spatial ones.

We implemented a prototype of MemSpate based on AFL++ [6] version 4.09a, and evaluated 12 widely used real-world open-source programs. We compared MemSpate with five state-of-the-art memory-related fuzzers: AFL++ [6], MemLock [17], UAFL [15], HTFuzz [20] and TortoiseFuzz [16]. The results demonstrate that MemSpate outperforms the other fuzzers in terms of discovering memory safety vulnerabilities. Specifically, MemSpate is able to detect 33.33%, 140.00%, 30.43%, 46.34% and 76.47% more vulnerabilities than AFL++, MemLock, UAFL, HTFuzz and TortoiseFuzz, respectively. Furthermore, MemSpate discovered 4 new previously unknown that had not been reported by any other studies.

In summary, this paper makes the following contributions.

1. We proposed a comprehensive memory usage protocol that addresses both memory temporal vulnerabilities and memory spatial vulnerabilities.
2. We designed and developed MemSpate, a grey-box fuzzer that utilizes a more comprehensive memory usage protocol to efficiently detect memory safety vulnerabilities.
3. We evaluated MemSpate on 12 real-world programs and compared it with five state-of-the-art memory-related fuzzers. The results demonstrate that MemSpate outperforms the other fuzzers in terms of discovering memory safety vulnerabilities. Furthermore, our experiments have led to the discovery of 4 previously unknown vulnerabilities.

2 Motivation

Listing 1 demonstrates a null pointer dereference vulnerability that was discovered by MemSpate. This vulnerability is present in the program *yasm*, due to the

Listing 1. A null-pointer-dereference in *yasm*

```

1  static MMacro *mmacros[NHASH];
2
3  static int expand_mmacro(Token *tline) {
4      Token **params, *t, *tt;
5      MMacro *m;
6      Line *l, *ll;
7      // ...
8      t = tline;
9      // ...
10     m = is_mmacro(t, &params); // get macro from defined table
11     // ...
12     for (l = m->expansion; l; l = l->next) {
13         // ...
14         for (t = l->first; t; t = t->next) {
15             Token *x = t;
16             if (t->type == TOK_PREPROC_ID && t->text[1] == '0' && t->text[2] == '0'
17                 ) // crash
18                 // ...
19             }
20         }
21     }
22
23     static MMacro *is_mmacro(Token *tline, Token ***params_arr) {
24         // ...
25         head = mmacros[hash(tline->text)];
26
27         for (m = head; m; m = m->next)
28             if (!strcmp(m->name, tline->text, m->casesense))
29                 break;
30         if (!m)
31             return NULL;
32         // ...
33         while (m) {
34             // ...
35             return m; // without checking every line is null
36             // ...
37         }
38     }

```

absence of validation for the pointer t before it is accessed. Specifically, within the file `nasm-preproc.c`, the function `expand_smacro()` utilizes a `Token` pointer t to process the text stored in `MMacro` m . However, m is initialized by the function `is_macro()` without validating the line during expansion before returning m . The function `expand_smacro()` only checks the type of token t , leading to a crash when attempting to access the line text of macro m . During our experiments, current fuzzing tools such as `AFL++`, `MemLock`, and `TortoiseFuzz` were unable to detect this vulnerability within 24 h. This is because they did not take the memory temporal information into account.

Another example is a heap buffer overflow vulnerability in the program `binutils`, as illustrated in Listing 2. Specifically, the array `shndx_pool`, with size `shndx_pool_size`, is initialized within the function `prealloc_cu_tu_list()`. However, when the function `add_shndx_to_cu_tu_entry()` attempts to write data to the array within the function `process_cu_tu_index()`, no bound checking is performed on the array, resulting in a heap buffer overflow. Similarly, existing

Listing 2. A heap-buffer-overflow in *binutils*

```

1  static void add_shndx_to_cu_tu_entry(unsigned int shndx) {
2  shndx_pool[shndx_pool_used++] = shndx; // out of bounds
3  }
4
5  static void prealloc_cu_tu_list(unsigned int nshndx) {
6  if (shndx_pool == NULL) {
7  shndx_pool_size = nshndx;
8  shndx_pool_used = 0;
9  shndx_pool = (unsigned int *)xcalloc(shndx_pool_size, sizeof(unsigned
10 int));
11 } else {
12 shndx_pool_size = shndx_pool_used + nshndx;
13 shndx_pool = (unsigned int *)xcrealloc(shndx_pool, shndx_pool_size,
14 sizeof(unsigned int));
15 }
16 }
17
18 static int process_cu_tu_index(struct dwarf_section *section, int do_display)
19 {
20 // ...
21 unsigned char *shndx_list;
22 unsigned int shndx;
23 // ...
24 if (!do_display) {
25 prealloc_cu_tu_list((limit - ppool) / 4);
26 for (shndx_list = ppool + 4; shndx_list <= limit - 4; shndx_list += 4) {
27 shndx = byte_get(shndx_list, 4);
28 add_shndx_to_cu_tu_entry(shndx); // entry
29 }
30 end_cu_tu_entry();
31 }
32 // ...
33 }

```

fuzzing tools such as AFL++, UAFL, and HTFuzz, which disregard memory spatial information, were incapable of detecting this vulnerability within 24 h.

Above all, existing fuzzing tools are unable to detect both of the aforementioned vulnerabilities. This is primarily because existing fuzzing solutions either overlook memory usage information or address it partially. As a result, there is a clear need for an enhanced fuzzer capable of identifying a broader spectrum of memory safety vulnerabilities in order to address this limitation.

3 Our Approach

3.1 Overview of MemSpate

The workflow of MemSpate is shown in Fig. 1, which consists of two main components: *static analysis* and *fuzzing loop*. MemSpate follows the general workflow of grey-box fuzzers but integrates improvements in both components guided by a memory usage protocol. In particular, the static analysis takes the program source code as the input, and generates the information related to the memory usage protocol, including control flow graph, call graph, memory operations, and memory sequences. Similar to the general grey-box fuzzers, the control flow graph information is utilized to gather the branch coverage, and the call graph

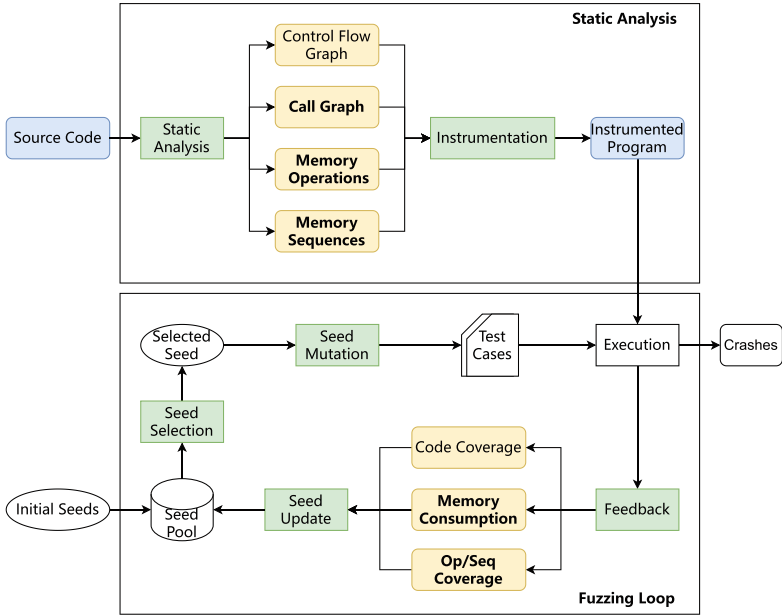


Fig. 1. Workflow of MemSpate

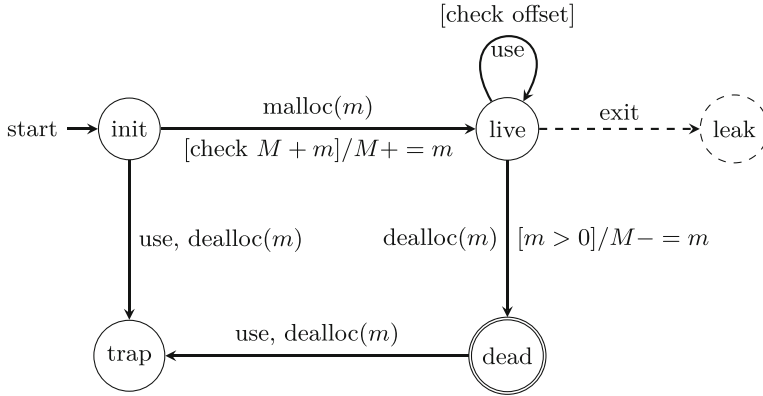
information is used to instrument the function call entries and returns. With guidance from the memory usage protocol, MemSpate identifies the locations of memory operations, calculates the (maximum) memory consumption, and analyzes the operation sequences that may violate the memory usage protocol. This information helps to determine where to instrument and what instrumentation is needed. Once the program is instrumented, MemSpate feeds it into the fuzzing loop for detecting memory safety vulnerabilities.

During the fuzzing loop, in addition to collecting the code coverage information, MemSpate also gathers information on memory consumption, memory operation coverage, and memory sequence coverage. MemSpate preserves the test cases that actively contribute to covering new code branches, consuming more memory, or covering new memory operation sequences, thus enhancing the likelihood of uncovering memory safety vulnerabilities.

3.2 Memory Usage Protocol

To detect a broader range of memory safety vulnerabilities, MemSpate employs a more comprehensive protocol to guide the fuzzing process.

The memory usage protocol used in MemSpate is represented as an automaton, which is illustrated in Fig. 2, where the conditions enclosed in square brackets denote the guards that the operation should satisfy, and if so, the corresponding action is performed. The protocol contains three kinds of memory operations, that is, memory allocation (e.g., malloc, new), memory deallocation (e.g., free,



M : Memory consumption
 m : Allocated/deallocated memory size

Fig. 2. Memory Usage Protocol of MemSpatе

delete) and memory use (e.g., read, write). And for both memory allocation and deallocation, a parameter m , representing the size of allocated/deallocated memory, is associated. To describe the effects of the memory operations, MemSpatе uses four states, that is, *init*, *live*, *dead*, *trap*, and *leak*. MemSpatе also uses a global variable M to represent the current memory consumption. When a request for memory allocation of size m arises, MemSpatе checks whether the sum of the current memory consumption M and the required memory size m exceeds a preset memory threshold (*i.e.*, the total system memory size). If this limit is exceeded, MemSpatе may report a warning regarding potential *memory overflow* or *allocation failure*. Conversely, if there is sufficient available memory, the current memory consumption M is updated as $M + m$, and a new memory block is allocated with its state marked as *live* (transitioning from *init*). This allocated memory block can be used as long as the accessed points (offsets) fall within its defined bounds. And it remains in *live* until either it is deallocated or the program exits. In the latter case, a potential *memory leak* would be reported, as indicated by the dotted state in Fig. 2). While in the first case, this memory block becomes *dead* and then the current memory consumption is reduced to $M - m$. However, if any use or deallocation is performed on an already-deallocated memory block, then it goes into *trap* state, which indicates the detection of a *use-after-free* vulnerability or a *double-free* vulnerability. Similarly, if any use or deallocation is performed on an uninitialized memory address (in *init* state), it indicates the detection of a *null-pointer-dereference* vulnerability or an *invalid-free* vulnerability.

Function Call Stack. In addition to addressing heap memory overflow vulnerabilities, MemSpatе also considers stack overflow vulnerabilities. Similarly, MemSpatе utilizes a global variable D to represent the current stack depth.

Algorithm 1: Memory Operation Sequences Analysis

Input: Original Program P
Output: Memory Operation Sequences Set Seq

- 1 $Seq \leftarrow \emptyset$;
- 2 $S_M \leftarrow findMalloc(P)$;
- 3 $S_U \leftarrow findUsage(P)$;
- 4 $S_F \leftarrow findDealloc(P)$;
- 5 **foreach** $(s_m, m) \in S_M$ **do**
- 6 $A \leftarrow getAlias(m)$;
- 7 $A_F \leftarrow findDealloc(A, P)$;
- 8 $A_U \leftarrow findUsage(A, P)$;
- 9 $Seq \leftarrow Seq \cup reachAnalysis(s_m, A_F, A_U)$;
- 10 $Seq \leftarrow Seq \cup reachAnalysis(s_m, A_F, A_F)$;
- 11 $Seq \leftarrow Seq \cup checkOffset(s_m, A_U)$;
- 12 $S_U \leftarrow S_U - A_U$;
- 13 $S_F \leftarrow S_D - A_F$;
- 14 $Seq \leftarrow Seq \cup S_U$ // **null-pointer-dereference**
- 15 $Seq \leftarrow Seq \cup S_F$ // **invalid-free**
- 16 **return** Seq ;

Upon function invocation, the variable D is incremented by 1; upon function return, D is decremented by 1. It is worth noting that recursive functions may be invoked excessively, gradually consuming the stack until it overflows.

3.3 Static Analysis

According to the memory usage protocol, any memory operation may introduce vulnerabilities if performed improperly (*i.e.*, violating the temporal rule of the protocol) or if cumulative memory consumption exceeds an acceptable limit (*i.e.*, violating the spatial rule of the protocol). Therefore, MemSpate will identify all potential memory operations within a program, which can be done during instrumentation. Furthermore, certain vulnerabilities arise from special memory operation sequences, such as use-after-free and double-free. For that, MemSpate will gather these memory operation sequences, particularly those with a length greater than 1, that result in the *trap* state.

Memory Operation Sequences. Inspired by UAFL [15], MemSpate conducts static analysis to identify the memory operation sequences that violate the temporal rule of the protocol. Algorithm 1 illustrates the basic idea, which takes a program P as input and outputs a set of memory operation sequences S .

To begin with, MemSpate gathers all memory allocation operations along with their corresponding memory objects in S_M (line 2). It also collects all memory use operations in S_U (line 3) and all memory deallocation operations in S_F (line 4). For each memory operation s_m and its associated object m , MemSpate employs pointer analysis to identify their potential aliasing pointers (line 6). Subsequently, MemSpate identifies all memory deallocation operations

Algorithm 2: Instrumentation

Input: Original Program P
Output: Instrumented Program P'

```

1 foreach function  $f \in P$  do
2   foreach basic block  $bb \in CFG_f$  do
3     if  $isEntryBB(bb)$  then
4        $stack\_depth \leftarrow stack\_depth + 1;$ 
5        $max\_stack\_depth \leftarrow max(max\_stack\_depth, stack\_depth);$ 
6     foreach instruction  $i \in bb$  do
7       if  $isReturnInst(i)$  then
8          $stack\_depth \leftarrow stack\_depth - 1;$ 
9       if  $isAllocInst(i)$  then
10         $size \leftarrow calculate\_size(i);$ 
11         $alloc\_size \leftarrow alloc\_size + size;$ 
12         $max\_alloc\_size \leftarrow max(max\_alloc\_size, alloc\_size);$ 
13      if  $isDeallocCond(i)$  then
14        if  $isDeallocInst(i)$  then
15           $size \leftarrow lookup\_size(i);$ 
16           $alloc\_size \leftarrow alloc\_size - size;$ 
17        foreach sequence  $seq \in getDeallocSequences(bb)$  do
18          if  $isLast(seq, bb)$  then
19             $op\_seqs[seq \oplus bb] \leftarrow op\_seqs[seq \oplus bb] + 1;$ 
20          else
21             $pos \leftarrow findDeallocPos(bb, seq);$ 
22             $op\_seqs[pos \oplus seq] \leftarrow op\_seqs[pos \oplus seq] + 1;$ 
23      if  $isUseCond(i) \vee isUseInst(i)$  then
24        foreach sequence  $seq \in getUseSequences(bb)$  do
25           $op\_seqs[seq \oplus bb] \leftarrow op\_seqs[seq \oplus bb] + 1;$ 
26     $code\_cov[bb_{pre} \oplus bb] \leftarrow code\_cov[bb_{pre} \oplus bb] + 1;$ 

```

that deallocate the memory object m via an aliasing pointer, yielding a set A_F (line 7). Similarly, MemSpate finds all memory use operations related to any aliasing pointer of m , yielding a set A_U (line 8). With the assistance of reachability analysis, MemSpate adds into Seq the paths of the operation sequence $[s_m, s_f, s_u]$ if s_m can reach s_f and s_f can reach s_u . As well as the paths of the operation sequence $[s_m, s_f, s'_f]$ if s_m can reach s_f and s_f can reach s'_f , where s_u is a use operation from S_U while s_f and s'_f are two different deallocation operations from S_F (lines 9–10). Additionally, if feasible, it also checks the offset is appropriate for use operations (line 11). After that, MemSpate respectively remove the operations in A_F and A_U from S_F and S_U (lines 12–13). Finally, MemSpate appends the remaining deallocation and usage operations into Seq (lines 14–15) and returns Seq (line 16).

Instrumentation. In order to monitor memory usage information and adjust the fuzzing strategy in the fuzzing loop accordingly, we perform instrumentation to collect both code coverage information and memory-related information. Algorithm 2 illustrates the process of program instrumentation.

MemSpate follows the standard workflow of AFL [9] and AFL++ [6] for handling code coverage: MemSpate instruments every basic block in the program using *code_cov* (line 26).

In terms of stack consumption, MemSpate instruments the entry (line 3) and return instructions (line 7) of each function. It respectively increments (line 4) and decrements (line 8) the stack depth by 1 correspondingly. Furthermore, MemSpate keeps track of the maximum stack depth (line 5), which is beneficial for seed selection during the fuzzing loop. Concerning heap consumption, MemSpate instruments each allocation operation (line 9) and each deallocation operation (line 13).

MemSpate calculates the memory size *size* and increases the memory consumption *alloc_size* by *size* for each allocation operation (lines 10–11); while MemSpate looks up the *size* and decreases the memory consumption *alloc_size* by *size* for each deallocation (lines 14–15). Similarly, MemSpate keeps track of the maximum memory consumption (line 12), which is beneficial for seed selection during the fuzzing loop.

Finally, MemSpate utilizes a bitmap, *op_seqs*, to track the memory operation sequence coverage at the basic block level, as generated by Algorithm 1. As previously discussed, the operations in sequences may necessitate certain path conditions (*i.e.*, the branch basic blocks required by the paths). If an instruction is deallocated with its path conditions holding (line 14), MemSpate retrieves all the sequences associated with the deallocation (line 17). For each sequence *seq*, if the deallocation appears last in *seq* (*i.e.*, a double-free sequence), MemSpate identifies and marks the sequence as covered (lines 18–19); otherwise (*i.e.*, a subsequence of a use-after-free or double-free sequence), MemSpate locates the position of the deallocation in *seq* and marks that position as covered (lines 21–22). The use operation follows similar procedures (lines 23–25).

3.4 Fuzzing Loop

The fuzzing loop of MemSpate is outlined in Algorithm 3. It takes the instrumented program P' and a set of initial seeds T as inputs and returns a set of test cases that trigger crashes S and a set of test cases that trigger memory safety vulnerabilities S_{mem} .

MemSpate initializes the seed pool *Queue* as the initial seeds T (line 3). Then MemSpate performs the following process until timeout: it selects a seed *seed* from the seed pool *Queue* (line 5) and assigns the seed an energy value *testcase_num* (line 6), which determines the number of children (*i.e.*, test-cases) to be generated from that seed (line 8), following the same heuristics as AFL++ [6]. After that, for each mutated testcase, MemSpate monitors the execution of the instrumented program P' and collects the information on code

Algorithm 3: Fuzzing Loop

Input: Instrumented Program P' , Initial Seed T **Output:** Crashes Set S , Memory Safety Vulnerabilities Set S_{mem}

```

1  $S \leftarrow \emptyset$ ;
2  $S_{mem} \leftarrow \emptyset$ ;
3  $Queue \leftarrow T$ ;
4 while  $time \leq timeout$  do
5    $seed \leftarrow selectSeed(Queue)$ ;
6    $testcase\_num \leftarrow assignEnergy(seed)$ ;
7   for  $i \leftarrow 1$  to  $testcase\_num$  do
8      $testcase_i \leftarrow mutate(seed)$ ;
9      $code\_cov_i, op\_seqs_i, max\_stack\_depth_i, max\_alloc\_size_i \leftarrow$ 
10     $fuzzRun(testcase_i, P')$ ;
11    if  $triggerCrash(testcase_i)$  then
12       $S \leftarrow S \cup testcase_i$ ;
13      if  $triggerMemCrash(testcase_i)$  then
14         $S_{mem} \leftarrow S_{mem} \cup testcase_i$ ;
15      if  $hasNewCov(op\_seqs_i)$  then
16         $Queue \leftarrow Queue \cup testcase_i$ ;
17      else if  $hasNewCov(code\_cov_i)$  then
18         $Queue \leftarrow Queue \cup testcase_i$ ;
19      else if  $isLarger(max\_alloc\_size_i, max\_stack\_depth_i)$  then
20         $Queue \leftarrow Queue \cup testcase_i$ ;

```

coverage, memory operation sequence coverage, maximum stack depth, and maximum memory consumption (line 9). If the program crashes, then the testcase is added to the crash set S (lines 10–11). Moreover, if the crash is classified as a memory-related vulnerability by sanitizers, then the testcase is added into set S_{mem} as well (lines 12–13). Otherwise, if the testcase is considered to be interesting, meaning that it achieves new code branch coverage (line 14), introduces new memory operation sequence coverage (line 16), or results in larger stack/heap memory consumption (line 18), then the testcase is added into the seed pool for further testing (lines 15, 17, 19).

4 Evaluation

We have implemented a prototype of our memory usage protocol guided fuzzer MemSpate based on AFL++ [6] version 4.09a, wherein SVF [14], a static value-flow tool, is used to implement the static analysis. Our main focus is on modifying the influencing factors in the instrumentation and feedback mechanism. By making these modifications without changing other components, we have successfully improved the overall performance of memory-related vulnerability detection.

We conducted comprehensive experiments to evaluate MemSpate using a set of real-world programs, and compared MemSpate with state-of-the-art fuzzers. In the experiments, we aim to answer the following research questions:

- RQ1.** How effective is MemSpate in detecting memory safety vulnerabilities in real-world programs?
- RQ2.** How does MemSpate compare to other state-of-the-art fuzzers?
- RQ3.** Can the protocol of MemSpate assist in detecting memory safety vulnerabilities more comprehensively?

4.1 Experiment Setup

Benchmark Programs. We curated a collection of 12 benchmark applications from fuzzing papers focusing on memory safety vulnerabilities, as shown in Table 1.

Table 1. Real-world programs evaluated in our experiment

No.	Program	Version	LoC	Input Format	Test Instruction
1	bento4-640	1.6.0-640	106K	mp4	mp42hls @@
2	bento4-639	1.6.0-639	105K	mp4	mp42hls @@
3	binutils	2.40	4984K	elf	readelf -w @@
4	cflow-1.7	1.7	91K	c	cflow @@
5	cflow-1.6	1.6	80K	c	cflow @@
6	cxxfilt	2.40	4984K	text	cxxfilt -t
7	giflib	5.2.1	17K	gif	gif2rgb @@
8	mjs	9eae0e6	49K	js	mjs -f @@
9	openh264	8684722	141K	text	h264dec @@ ./tmp
10	yara	3.5.0	63K	text	yara @@ strings
11	yaml-cpp	0.6.2	122K	text	parse @@
12	yasm	9defefa	176K	asm	yasm @@

Baseline Fuzzers. We evaluated MemSpate by comparing it against five state-of-the-art fuzzers: AFL++ [6], MemLock [17], UAFL [15], HTFUZZ [20] and TortoiseFuzz [16]. These fuzzers were selected based on several factors. Firstly, AFL++ is an improved version of AFL [9] and has established itself as one of the most widely used baselines in recent research papers. Both Memlock and TortoiseFuzz have proposed strategies for memory operations to discover memory spatial vulnerabilities, while UAFL and HTFUZZ focus on detecting memory temporal vulnerabilities via feedback on memory operation sequences. Since UAFL is not publicly available, we made efforts to replicate it in our environment and refer to our replication of UAFL as UAFL[†].

Evaluation Metrics. Since most baseline fuzzers and MemSpate focus on detecting memory safety vulnerabilities, we evaluate the performance of the fuzzers in terms of the number of memory safety vulnerabilities, instead of the number of unique crashes as traditional coverage-guided grey-box fuzzers do. Similar to TortoiseFuzz [16] and HTFuzz [20], we utilized AddressSanitizer [13] to analyze the crashes and to identify memory safety vulnerabilities. Additionally, we manually reviewed the crash reports to identify unique vulnerabilities and compared them with existing CVEs and GitHub issues to discover new vulnerabilities.

Experiment Configuration. Each experiment ran for 24 h, and the command options for the benchmark programs are listed in the last column of Table 1. According to Klees’s suggestions [8], each experiment was conducted 10 times to minimize the influence of randomness.

Experiment Infrastructure. All the experiments were conducted using the same setup: a docker container configured with 1 CPU core of Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz and the 64-bit Ubuntu 18.04 LTS.

Table 2. Number of memory safety vulnerabilities found by different fuzzers

Program	MemSpate		AFL++		MemLock		UAFL†		TortoiseFuzz		HTFuzz	
	Uniq	Avg	Uniq	Avg	Uniq	Avg	Uniq	Avg	Uniq	Avg	Uniq	Avg
bento4-640	1	0.10	1	0.10	0	0.00	0	0.00	0	0.00	1	0.10
bento4-639	4	4.00	4	4.00	4	2.50	4	3.80	4	2.60	4	3.80
binutils	1	0.30	0	0.00	0	0.00	1	0.10	0	0.00	1	0.10
cflow-1.7	4	1.60	1	0.80	3	1.10	1	0.80	1	0.30	1	0.50
cflow-1.6	4	2.80	4	1.90	3	1.80	5	2.20	3	1.30	2	1.10
cxxfilt	1	0.20	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
giflib	1	0.50	1	0.30	1	0.10	0	0.00	1	0.40	1	0.10
mjs	12	5.50	7	4.90	3	2.10	9	5.50	7	4.90	7	5.20
openh264	2	1.10	1	1.00	0	0.00	1	1.00	1	0.90	1	1.00
yaml-cpp	5	2.70	5	2.40	6	3.00	5	2.60	5	2.30	0	0.00
yara	9	4.00	6	3.50	3	1.90	5	3.40	5	3.10	6	5.30
yasm	16	10.90	15	8.00	2	0.70	15	9.70	14	9.70	10	7.70
total	60	33.70	45	26.90	25	13.20	46	29.10	41	25.50	34	24.90

4.2 Memory-Related Vulnerability Detection Capability (RQ1)

The “MemSpate” column in Table 2 presents the results of MemSpate in detecting memory safety vulnerabilities, where the term “Uniq” denotes the total number of unique vulnerabilities found during the 10 runs while “Avg” denotes the average number of unique vulnerabilities among the 10 runs. As depicted in

Table 2, MemSpate successfully found a total of 60 memory safety vulnerabilities and an average of 33.70 ones across the 12 benchmark programs. Notably, MemSpate was able to identify more than 10 memory safety vulnerabilities in *mjs* and *yasm*.

Moreover, we manually reviewed the vulnerabilities and compared them with existing CVEs and GitHub issues. And we found that MemSpate is able to find 4 previously unknown memory safety vulnerabilities in *yasm*, which have been submitted in GitHub and are listed in Table 3.

Table 3. New memory safety vulnerabilities found by MemSpate

Program	Version	Report	Vulnerability Type
yasm	9defefa	Issue-273	null-pointer-dereference
		Issue-274	heap-use-after-free
		Issue-276	heap-buffer-overflow
		Issue-277	null-pointer-dereference

According to the above results, we conclude that MemSpate is capable of detecting memory safety vulnerabilities in real-world programs.

4.3 Comparison with Other Memory-Related SOTA Fuzzers (RQ2)

Table 2 also presents the number of memory safety vulnerabilities detected by each baseline fuzzer (e.g., AFL++, MemLock, UAFL†, TortoiseFuzz, and HTFuzz) on 12 benchmark programs over 24 h, where the numbers in bold indicate that the corresponding fuzzers achieve the best results. The results demonstrate that MemSpate achieves superior performance in terms of both total unique vulnerability number and average vulnerability number. Specifically, compared with AFL++, MemLock, UAFL†, TortoiseFuzz and HTFuzz, MemSpate can respectively identify approximately 33.33%, 140.00%, 30.43%, 46.34% and 76.47% more unique vulnerabilities in total, as well as respectively identifying 25.82%, 155.30%, 15.81%, 32.16% and 35.34% more vulnerabilities on average. Moreover, MemSpate demonstrates superior performance across most programs. In particular, on the program *mjs*, MemSpate identifies 12 memory safety vulnerabilities, which is significantly higher compared to any other fuzzers.

Figure 3 illustrates the trend of memory safety vulnerabilities found by each fuzzer over time. Within the initial 6-h period, all baseline fuzzers have reached a convergence point and successfully detected over 75.00% of the vulnerabilities within their detection scope. However, for MemSpate, 36.67% of detected vulnerabilities are explored in the later stage, resulting in a significantly higher number of detected vulnerabilities compared to the baseline fuzzers.

Based on the findings presented in Table 2, we have computed the p-value of the Mann-Whitney U-test between MemSpate and each baseline fuzzer. The

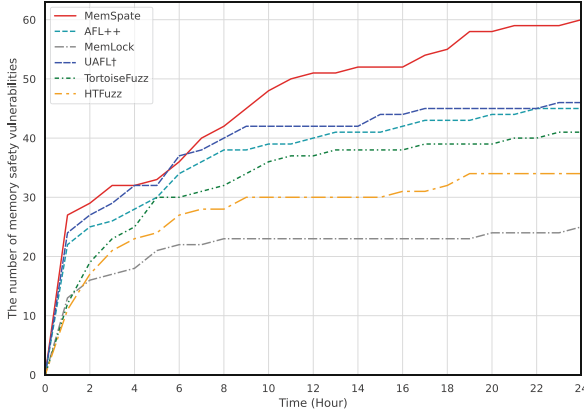


Fig. 3. Trend of memory safety vulnerabilities found by each fuzzer over time

Table 4. P-values of memory safety vulnerabilities found in 10 runs

programs	AFL++	MemLock	UAFL†	TortoiseFuzz	HTFuzz
bento4-640	5.29e-01	1.84e-01	1.84e-01	1.84e-01	5.29e-01
bento4-639	1.00e+00	1.01e-04	8.37e-02	9.86e-05	8.37e-02
binutils	3.84e-02	3.84e-02	1.50e-01	3.84e-02	1.50e-01
cflow-1.7	1.81e-03	1.26e-02	1.81e-03	2.45e-04	6.77e-04
cflow-1.6	1.15e-03	8.56e-03	5.06e-03	8.92e-05	6.89e-05
cxxfilt	8.37e-02	8.37e-02	8.37e-02	8.37e-02	8.37e-02
giflib	1.99e-01	3.18e-02	6.83e-03	3.47e-01	3.18e-02
mjs	1.15e-01	3.95e-05	6.22e-01	1.15e-01	2.64e-01
openh264	1.84e-01	1.21e-05	1.84e-01	9.58e-02	1.84e-01
yaml-cpp	1.02e-01	9.33e-01	3.03e-01	4.46e-02	2.02e-05
yara	8.09e-02	1.63e-04	6.63e-02	8.60e-03	9.99e-01
yasm	7.42e-04	5.97e-05	4.00e-02	4.77e-02	5.97e-05

outcomes are shown in Table 4, with values highlighted in bold indicating significance levels below 0.05. Our analysis reveals that MemSpate outperforms all the five compared fuzzers in 29 out of the 60 comparisons with a significant difference.

Overall, our experimental results demonstrate that MemSpate outperformed AFL++, MemLock, UAFL†, TortoiseFuzz, and HTFuzz in identifying memory safety vulnerabilities.

4.4 Effectiveness of Memory Usage Protocol (RQ3)

Figure 4 presents the number of memory safety vulnerabilities categorized by various types found by each fuzzer. The findings demonstrate that MemSpate

is capable of detecting all 9 types of memory safety vulnerabilities, whereas other fuzzers fail to detect 2 or 3 types among the 12 benchmark programs. Furthermore, MemSpat has identified a total of 33 memory temporal vulnerabilities (including double-free, heap-use-after-free, invalid-free and null-pointer-dereference) and 27 memory spatial vulnerabilities (the others). In comparison, AFL++, MemLock, UAFL†, TortoiseFuzz and HTFuzz have found 27, 8, 28, 25 and 21 memory temporal vulnerabilities respectively. Additionally, they have identified 18, 17, 18, 16 and 13 memory spatial vulnerabilities. Notably, only MemSpat has the ability to uncover the invalid-free vulnerability.

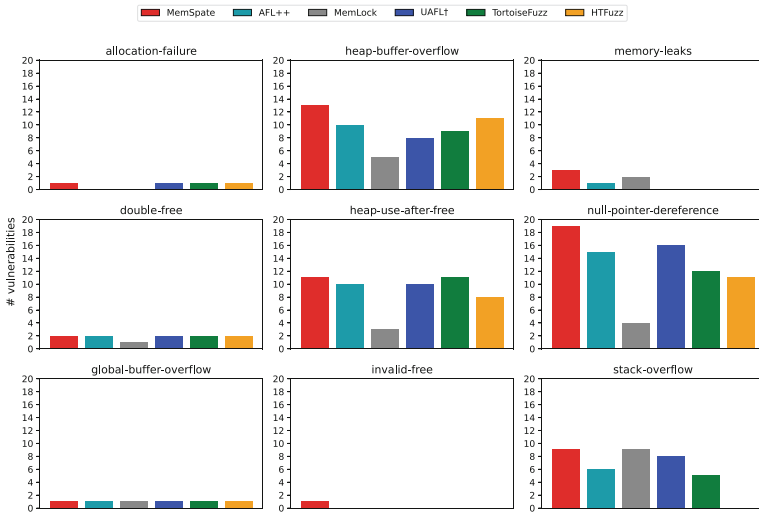


Fig. 4. Vulnerability numbers categorized by various types

Figure 5 presents upset plots illustrating the collective number of vulnerabilities discovered by different sets of fuzzers. The figure indicates that there exists a total of 12 vulnerabilities that can be found by all the fuzzers. Furthermore, MemSpat demonstrates the capability to exclusively identify 8 vulnerabilities, while both UAFL† and TortoiseFuzz can each uniquely identify one vulnerability. Among all the vulnerabilities, MemSpat only missed 6 vulnerabilities, whereas AFL++, MemLock, UAFL†, TortoiseFuzz, and HTFuzz missed a greater number at 21, 41, 20, 25, and 32, respectively.

Based on the above findings, we can deduce that the memory usage protocol of MemSpat contributes to a more comprehensive detection of memory safety vulnerabilities, encompassing both a greater number and variety of types.

4.5 Discussion

Overhead of Instrumentation. This paper presents our proposed fuzzer, MemSpat, which effectively detects memory safety vulnerabilities. However,

due to the high expressiveness of preliminary static analysis, the instrumentation component introduces more overhead to the target program compared to other fuzzers. We calculated the average time (in seconds) spent on 10 runs of instrumentation, and the results are shown in Table 5.

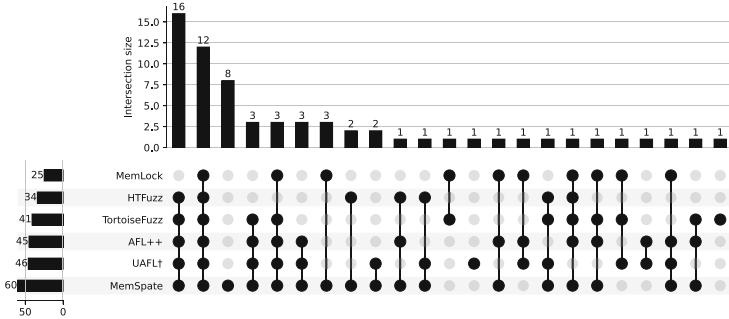


Fig. 5. UpSet Plot for MemSpate and five baseline fuzzers

It is evident that MemSpate consumes an average of 517.32s to instrument the target programs, which is significantly higher than its base fuzzer AFL++ (3.74 ×). Considering our replication UAFLL† of UAFLL on MemSpate, we find that its average cost of instrumentation (3.70 × compared to AFL++) is slightly lower than that of MemSpate. MemLock, TortoiseFUZZ, and HTFUZZ are built on top of AFL and may have a lower instrumentation overhead than AFL++. While MemLock’s instrumentation is even slightly lower than AFL++ (0.90 ×), TortoiseFUZZ and HTFUZZ still have a higher instrumentation overhead than AFL++ (1.96 × and 2.04 ×) due to the complex memory safety protocol they propose; however, both are far lower than MemSpate and UAFLL†.

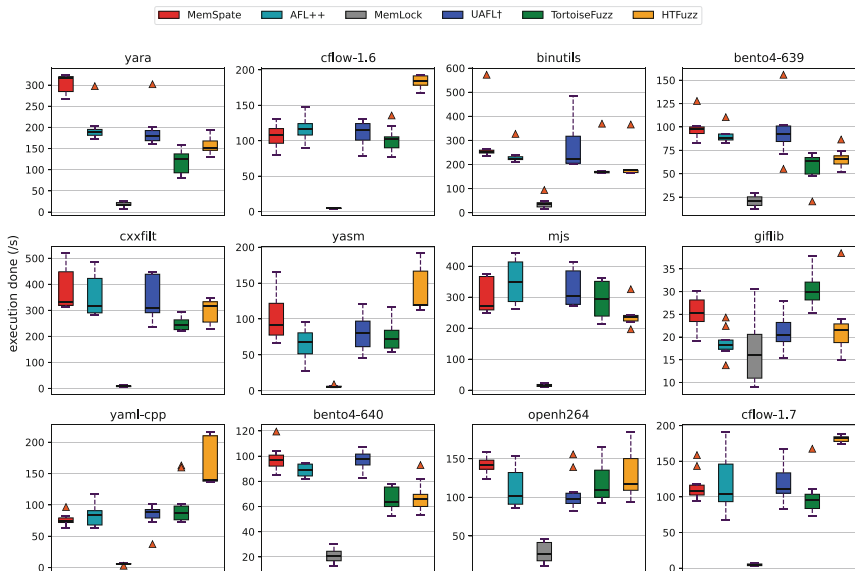
Upon thorough analysis, we assert that the high instrumentation overhead of MemSpate, along with the requirement for static analysis and additional pre-processing before conducting static analysis, also contributes to a considerable expense that cannot be ignored.

Moreover, an experiment was conducted to investigate the impact of our instrumentation on the fuzzing loop phase. The speed of the fuzzing loop for each fuzzer on each benchmark program was measured in terms of executions per second. Figure 6 illustrates the results from 10 runs. The findings indicate that the executions per second of MemSpate are not significantly lower than those of the other base fuzzers and even show significant improvement compared to the other base fuzzers on program *yara*.

In conclusion, the instrumentation cost of MemSpate in the static analysis stage results in a significant time investment. However, the time overhead of these static analyses (approximately 10 min) falls within acceptable limits when compared to the duration of the fuzzing loop test (24 h as we have set). Furthermore, the impact of instrumentation on the efficiency of fuzzing loop execution is negligible.

Table 5. Instrumentation overhead of fuzzers

programs	MemSpate	AFL++	MemLock	UAFL†	TortoiseFuzz	HTFuzz
bento4-639	466.90	182.10	140.40	436.00	174.40	152.00
bento4-640	461.10	193.10	133.90	454.10	185.40	150.10
binutils	1275.10	452.10	407.10	1285.10	907.40	867.30
cflow-1.6	243.70	42.10	44.10	245.50	65.20	58.80
cflow-1.7	266.10	47.80	48.80	285.50	74.30	64.20
cxxfilt	1865.60	449.50	432.00	1888.70	892.10	831.30
giflib	90.60	33.50	30.40	90.70	35.10	31.90
mjs	36.00	5.20	4.80	32.80	12.30	10.90
openh264	134.60	39.10	37.20	117.30	145.40	114.40
yaml-cpp	387.40	133.70	139.50	342.00	607.10	966.70
yara	457.10	33.70	32.90	470.10	58.20	51.70
yasm	523.60	47.50	41.60	490.60	93.30	81.90
Average	517.32	138.28	124.39	511.53	270.85	281.77

**Fig. 6.** Executions per second of each fuzzer in 10 runs

Threats to Validity. We discuss the potential threats to the validity and generalizability of our study, as well as the measures we have taken to mitigate or control them. One potential threat is the selection bias that may arise from using only 12 open-source programs, which could limit the diversity of our dataset. To address this concern, we made sure to select diverse programs from vari-

ous domains and with different characteristics. We are continuously working on improving and evaluating MemSpate. Another potential concern entails the sampling error that may arise when utilizing a restricted number of seeds and inputs for each program and fuzzer. This limitation has the potential to impact the comprehensive nature of our testing process and introduce factors that create noise or variance in our findings. To address this, we employed an identical set of seeds for each fuzzer and conducted a 24-h runtime. We repeated each experiment 10 times and reported the average and standard deviation to account for any potential variations. A third threat to consider is the possibility of statistical errors. In order to compare MemSpate with other testers, we utilized statistical tests and significance levels. However, it is important to acknowledge that these tests have certain assumptions and limitations. To address this concern, we thoroughly checked the assumptions and conditions before applying the tests, ensuring that we used the appropriate test for each specific scenario.

5 Related Work

There are various memory-related grey-box fuzzing solutions, whose target can be classified into two categories: memory spatial bugs and memory temporal bugs.

Fuzzing for Memory Spatial Bugs. Dowser [7], SAFAL [2] and a concolic execution-based smart fuzzing method [11] are specifically designed to detect buffer overflow vulnerabilities. MemFuzz [3] leverages information about memory accesses to guide the fuzzing process. MemLock [17] employs memory consumption information to guide the fuzzing process. MemConFuzz [4] extracts the locations of heap operations and data-dependent functions through static data flow analysis. ovAFLow [21] broadens the vulnerable targets to memory operation function arguments and memory access loop counts. TortoiseFuzz [16] proposes a new metric *coverage accounting* to evaluate coverage by security impacts (*i.e.*, memory operations), and introduces a new scheme to prioritize fuzzing inputs.

Fuzzing for Memory Temporal Bugs. FUZE [18] is a new framework to facilitate the process of kernel UAF exploitation. UAFuzz [12] relies on user-defined UAF sites to guide the fuzzer during exploration. UAFL [15] uses types-tate automata to describe a memory temporal protocol of use-after-free vulnerability. HTFuzz [20] only focuses on the temporal memory vulnerability. LTL-FUZZER [10] supports a linear-time temporal logic protocol, but requires expert knowledge to instrument program locations related to potential temporal violations. MDFuzz [23] identifies memory operation sequences as targets to guide the fuzzer without wasting resources exploring unrelated program components.

Note that existing fuzzing solutions typically only address memory usage partially, limiting their ability to detect all types of memory safety vulnerabilities. While MemSpate utilizes a more comprehensive memory usage, and thus is able to detect more memory safety vulnerabilities.

6 Conclusion

We have proposed MemSpate, a fuzzing technique that utilizes a more comprehensive memory usage protocol to efficiently detect memory safety vulnerabilities. We have evaluated MemSpate on 12 real-world programs, demonstrating its superior performance over 5 state-of-the-art fuzzers in terms of detecting memory safety vulnerabilities. Additionally, we have disclosed 4 previously unknown vulnerabilities to the respective vendors. This underscores MemSpate's efficacy in testing real-world programs that are prone to memory safety vulnerabilities.

Acknowledgements. This work was supported in part by the National Natural Science Foundation of China (Nos. 62472339, 62372304, 62302375, 62192734), the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Fundamental Research Funds for the Central Universities.

References

1. Ba, J., Duck, G.J., Roychoudhury, A.: Efficient greybox fuzzing to detect memory errors. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–12 (2022)
2. Bhardwaj, M., Bawa, S.: Fuzz testing in stack-based buffer overflow. In: Advances in Computer Communication and Computational Sciences: Proceedings of IC4S 2017, vol. 1, pp. 23–36. Springer (2019)
3. Coppik, N., Schwahn, O., Suri, N.: Memfuzz: using memory accesses to guide fuzzing. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 48–58. IEEE (2019)
4. Du, C., Cui, Z., Guo, Y., Xu, G., Wang, Z.: Memconfuzz: memory consumption guided fuzzing with data flow analysis. *Mathematics* **11**(5), 1222 (2023)
5. Farkhani, R.M., Ahmadi, M., Lu, L.: {PTAuth}: temporal memory safety via robust points-to authentication. In: 30th USENIX Security Symposium (USENIX Security 21), pp. 1037–1054 (2021)
6. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: {AFL++}: combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20) (2020)
7. Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H.: Dowsing for {Overflows}: a guided Fuzzer to find buffer boundary violations. In: 22nd USENIX Security Symposium (USENIX Security 13), pp. 49–64 (2013)
8. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138 (2018)
9. LCAMTUF: American Fuzzy Loop (2017). <https://lcamtuf.coredump.cx/afl/>
10. Meng, R., Dong, Z., Li, J., Beschastnikh, I., Roychoudhury, A.: Linear-time temporal logic guided Greybox fuzzing. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1343–1355 (2022)
11. Mouzarani, M., Sadeghiyan, B., Zolfaghari, M.: A smart fuzzing method for detecting heap-based buffer overflow in executable codes. In: 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 42–49. IEEE (2015)

12. Nguyen, M.D., Bardin, S., Bonichon, R., Groz, R., Lemerre, M.: Binary-level directed fuzzing for {use-after-free} vulnerabilities. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020), pp. 47–62 (2020)
13. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: {AddressSanitizer}: A fast address sanity checker. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 309–318 (2012)
14. Sui, Y., Xue, J.: SVF: interprocedural static value-flow analysis in LLVM. In: Proceedings of the 25th International Conference on Compiler Construction, pp. 265–266. ACM (2016)
15. Wang, H., et al.: Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 999–1010 (2020)
16. Wang, Y., Jia, X., Liu, Y., Zeng, K., Bao, T., Wu, D., Su, P.: Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: NDSS (2020)
17. Wen, C., et al.: Memlock: memory usage guided fuzzing. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 765–777 (2020)
18. Wu, W., et al.: {FUZE}: towards facilitating exploit generation for kernel {Use-After-Free} vulnerabilities. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 781–797 (2018)
19. Ye, D., Su, Y., Sui, Y., Xue, J.: Wpbound: enforcing spatial memory safety efficiently at runtime with weakest preconditions. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering, pp. 88–99. IEEE (2014)
20. Yu, Y., et al.: Htfuzz: heap operation sequence sensitive fuzzing. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–13 (2022)
21. Zhang, G., Wang, P.F., Yue, T., Kong, X.D., Zhou, X., Lu, K.: ovaflow: detecting memory corruption bugs with fuzzing-based taint inference. *J. Comput. Sci. Technol.* **37**(2), 405–422 (2022)
22. Zhang, T., Lee, D., Jung, C.: Bogo: buy spatial memory safety, get temporal memory safety (almost) free. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 631–644 (2019)
23. Zhang, Y., Wang, Z., Yu, W., Fang, B.: Multi-level directed fuzzing for detecting use-after-free vulnerabilities. In: 2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 569–576. IEEE (2021)



The Continuum Hypothesis Implies the Existence of Non-principal Arithmetical Ultrafilters – A Coq Formal Verification

Guowei Dou^{1(✉)}, Si Chen¹, Wensheng Yu^{1(✉)}, and Ru Zhang²

¹ Beijing Key Laboratory of Space-Ground Interconnection and Convergence, School of Electronic Engineering, Beijing University of Posts and Telecommunications, Beijing 100876, China

{[dgw](mailto:dgw@bupt.edu.cn), [wsyu](mailto:wsyu@bupt.edu.cn)}@bupt.edu.cn

² School of Science, Beijing University of Posts and Telecommunications, Beijing 100876, China

Abstract. The formalization of mathematical theorems is an important direction in the field of formal verification. Formalizing mathematical theorems ensures their accuracy and rigor in practical applications. Non-principal arithmetical ultrafilter (NPAUF) was proposed in [27]. It can be directly applied to the extension of number systems such as real numbers and non-standard real numbers. So far, though the existence of NPAUF has not been proven only with general set theories, it can be proven with the help of some additional consistent set-theoretic assumptions, such as the Continuum Hypothesis (CH). This paper presents the formal verification of the existence of NPAUF, implemented with the Coq proof assistant and grounded in the Morse-Kelley (MK) axiomatic set theory augmented with CH. The formal descriptions for the concepts related to filter, arithmetical ultrafilter (AUF), NPAUF, and CH are all provided. This work serves as the first step of our long-term objective – to formalize the non-standard analysis.

Keywords: formalization · Coq · Morse-Kelley axiomatic set theory (MK) · filter · arithmetical ultrafilter (AUF) · non-principal arithmetical ultrafilter (NPAUF) · Continuum Hypothesis (CH)

1 Introduction

This paper is dedicated to the formal verification of a mathematical theory – the existence of non-principal arithmetical ultrafilters (NPAUF), which was mentioned by Wang in [27] then officially named in [28]. The formalization is implemented in the Coq proof assistant.

Ultrafilter is a concept derived from topology and is widely applied in logic, set theory, model theory, etc. [8] Ultrafilter can be divided into principal and non-principal ultrafilter, and NPAUF is exactly a special kind of the latter for the reason that

- its existence may not be proven within general axiomatic set theories [27, 29].
- it can be directly applied in the non-standard extensions of algebraic structures [27, 30].

Bartoszynski and Shelah once asserted that the existence of NPAUF cannot be proven solely within the framework of ZFC set theory [3]. Although this announcement was later retracted due to the discovery of errors in the proof, it is generally believed that the conclusion is likely still correct [29, 30]. Proving the existence of NPAUF necessitates seeking support from additional set theory hypotheses. Among the various set theory hypotheses capable of deducing NPAUF, the Continuum Hypothesis (CH) is undoubtedly well-known and regarded as a safe assumption [4, 7, 29, 30].

In [27], Wang mentioned a real number theory involving the use of NPAUF. With the aid of NPAUF, the set of natural numbers ω can be extended to the non-standard natural number set ${}^*\mathbf{N}$ that encompasses infinity natural numbers. Additionally, following classical methods of extending natural numbers to rational numbers, ${}^*\mathbf{N}$ can be further extended to the non-standard integer set ${}^*\mathbf{Z}$ inclusive of infinity integers, and the non-standard rational number set ${}^*\mathbf{Q}$, which encompasses both infinity and infinitesimal numbers. The structure of real numbers is derived by equivalently classifying a specific subset of ${}^*\mathbf{Q}$. This theory utilizes a non-standard extension approach to bypass the usual rational numbers and directly construct real numbers [30].

Following the thought of the process from ω to ${}^*\mathbf{Q}$, it is possible that Wang's method could be applied in the construction of hyper-real numbers (i.e., non-standard real numbers), which is the foundation of non-standard analysis. Since the birth of non-standard analysis, the construction of hyper-real numbers has mainly relied on ultrapower and ultraproduct method [15, 17, 22], without widely accepted alternatives. Constructing hyper-real numbers using NPAUF could be a new method in the framework containing CH, which is of significant importance for the future development of mathematics. Robinson once quoted Gödel's statements in the preface of his masterpiece *Nonstandard Analysis* [22]: "There are good reasons to believe that non-standard analysis, in some version or other, will be the analysis of the future."

Our long-term goal is to formally prove Wang's real number theory, and apply this method to the formalization of hyper-real numbers, based on which the formalization of non-standard analysis will be implemented. The formalization of NPAUF's existence is a minor section and the first step of our final objective.

The study of filters needs to be conducted within the context of set theories. In mathematical research, Zermelo-Fraenkel (ZFC) axiomatic set theory is most widely adopted, and there already are various research works involving the formalization of ZFC [2, 26, 32, 33, 36]. Another important axiomatic set theory is Morse-Kelley set theory (MK), which was first proposed by Wang in 1949 [31] and was published as the appendix in Kelley's *General Topology* [18] in 1955. It includes eight axioms, one axiom schema, and 181 definitions or theorems. Different from ZFC, MK acknowledges "classes" (which are more numerous than sets) as fundamental objects. That is to say, every mathematical object (ordered

pair, function, integer, etc.) is a class, and only those classes belonging to some other ones are defined as sets. The non-set classes are named “proper classes”. Monk, Rubin, and Mendelson submit that MK does what is expected of a set theory while being less cumbersome than ZFC [19,20,23]. In fact, ZFC can be proven consistent in MK [5]. We consider that MK is a proper extension of ZFC and is convenient to utilize in formalization processes [24,35]. We have finished the MK formalization in Coq [24,35], and the work of this paper is based on it.

Coq is an interactive theorem prover, which serves as a computer tool for describing definitions, theorems and verifying proofs [6,9,16,21]. Coq is one of the most widely used and well-regarded theorem provers today. The most remarkable achievements based on Coq include, but are not limited to: In 2008, Gonthier and Werner successfully provided a computer proof of the famous Four Color Theorem based on Coq [13]; subsequently, after years of effort, Gonthier and his colleagues completed the formal verification of the Odd Order Theorem in 2013 [14]. The renowned mathematician and computer scientist Wiedijk believes that the ongoing formalization of mathematics is a mathematical revolution [34]. “With the help of computational proof assistants, formal verification could become the new standard for rigor in mathematics.” [1]

This paper presents the formal description of the concepts of filter, ultrafilter, NPAUF and more, then details the formal verification of the existence of NPAUF. The work is implemented in Coq proof assistant and based on MK formalization. The mathematical definitions and theorems used in this paper are mainly taken from textbooks [18,29,30] ([18] contributes to the MK parts and [29,30] to the filter parts). The entire Coq code is available at [here](#).¹

The paper is organized as follows: Sect. 2 summarizes the existing formalization work on MK; Sect. 3 details auxiliary results about filters prepared for the formalization of NPAUF; Sect. 4 presents the entire formal proof of the existence of NPAUF; Sect. 5 is conclusion.

2 Morse-Kelley Axiomatic Set Theory

Our team have finished the formalization of Morse-Kelley (MK) axiomatic set theory [24,35]. We summarize its important content as preparatory knowledge.

MK is grounded in Classical Logic [18], whereas the Coq system adopts Intuitionistic Logic [6]. Therefore, the Law of Excluded Middle needs to be assumed:

```
Axiom classic :  $\forall (P : \text{Prop}), P \vee \sim P.$ 
```

Compared to ZFC that only pays attention to the objects of “set”, MK acknowledges the more extensive objects of “class”. The term “class” does not appear in any axiom, definition or theorem in MK, but the primary interpretation of these statements is as assertions about classes. In Coq, the term “class” is declared as a new type “Class”, which is living in the topmost sort “Type”.

```
Parameter Class : Type.
```

¹ <https://github.com/1DGW/Formal-verification-of-the-existence-of-non-principal-arithmetical-ultrafilters-in-Coq/releases/tag/v1.0>.

There are two primitive constants besides the term “class”. The first is “ \in ”, which is read “is a member of” or “belongs to”. This means that one mathematical object (class) can be an element of another one. The second is denoted, rather strangely, “ $\{ \cdot : \cdot \}$ ” and is read “the class of all \cdot such that \cdot ”. It is the classifier and represents a class consisting of classes that satisfy a specific property. For example, “ $\{x : x \notin y\}$ ” represents the class that consists of all members not belonging to y . The constant “ \in ” is described as “In” and “ $\{ \cdot : \cdot \}$ ” as “Classifier”.

```
Parameter In : Class -> Class -> Prop.
Parameter Classifier : (Class -> Prop) -> Class.
```

```
Notation "x ∈ y" := (In x y) (at level 70).
Notation "\{ P \}" := (Classifier P) (at level 0).
```

The type of “In” is actually “Class -> Class -> Prop”, signifying that it takes two classes as input and produces a proposition. For instance, given classes x and y , “ $x \in y$ ” represents the proposition that x belongs to y .

The type of “Classifier” is slightly more intricate. In its type “(Class -> Prop) -> Class”, the first “Class” corresponds to the first placeholder in the classifier constant, intended to be filled by a variable representing a member of the classifier. The “Prop” corresponds to the second placeholder, meant for a proposition (whether correct or not). The last “Class” indicates that the classifier is indeed a class, but we do not know if there are members in it. Taken together, “(Class -> Prop)” implies that the proposition in the second placeholder takes the variable from the first as a parameter. Thus, in the notation “ $\{ P \}$ ”, the type of “P”, which represents a proposition with a parameter, is precisely “Class -> Prop”.

The constants “class”, “ \in ” and “ $\{ \cdot : \cdot \}$ ” form the basic structure of MK, through which all the axioms, definitions and theorems in MK can be described.

The definition of “set” is one of the most crucial definitions in MK, it constrains the concept of set to avoid the Russell’s Paradox, excluding certain classes that are deemed “too large” (e.g., $\{u : u = u\}$, $\{u : u \notin u\}$) [18, 24, 35].

Definition (Set). x is a set if and only if for some y , $x \in y$.

```
Definition Ensemble x := ∃ y, x ∈ y.
```

A class that is a member of another class is termed a set. A class that is not a set is called a proper class.

MK inherits most elementary concepts and operations concerning sets (e.g., intersection, union, complement, pairs) from naive set theory and extends them to all classes. For example, $x \cap y$ in MK represents the intersection of classes x and y , either of which could be a set or not. The table presented in Appendix A lists the fundamental definitions and mathematical meanings in MK along with their notations in Coq, which will be frequently used in latter sections.

Additionally, the entire MK code attached to this paper differs somewhat from the version we presented in [24, 35], with the main changes being: separating all the axioms and definitions of MK into one file as its main structure, and theorems along with their proofs into another file for easy lookup and calling.

This structured presentation makes the entire MK formalization concise and readable. For more details about MK, refer to [18,24,35]; or access Coq-related resources of MK from [here](#).²

3 Auxiliary Results

As previously mentioned in the [Introduction](#), to prove the existence of NPAUF, concepts about filters are involved. This section introduces our formalization work on filter-related concepts that will be used in the formal proof.

3.1 About Filters

Definition 1 (Filter Base). *For each set A, assume that B is a non-empty family of subsets of A (i.e., $B \neq \emptyset \wedge B \subset 2^A$) and satisfies:*

- 1) $\emptyset \notin B$,
- 2) if $a, b \in B$, then $a \cap b \in B$.

where $2^A (= \{u : u \subset A\})$ is the power set of A, and $B \subset 2^A$ represents that B is a subset of 2^A (not strict). Then B is called a filter base over A.

The formalization of filter base requires two parameters B and A. Besides, A and B have type “Class”, which means that the formal definition is more general, applicable to any classes.

Definition FilterBase B A := B <> Φ \wedge B \subset pow(A)
 $\wedge \Phi \notin B \wedge (\forall a b, a \in B \rightarrow b \in B \rightarrow (a \cap b) \in B)$.

where “pow(A)” represents the power class of A, “<>” represents inequality sign, and “ Φ ” the empty set. The interpretations of other notations here are same as those in mathematics.

Definition 2 (Filter). *For each set A, assume that F is a family of subsets of A (i.e., $F \subset 2^A$) and satisfies:*

- 1) $\emptyset \notin F, A \in F$,
- 2) if $a, b \in F$, then $a \cap b \in F$,
- 3) if $a \subset b \subset A$ and $a \in F$, then $b \in F$.

F is called a filter over A.

Just similar to the formalization of filter base, two parameters F and A are required.

Definition Filter F A := F \subset pow(A) $\wedge \Phi \notin F \wedge A \in F$
 $\wedge (\forall a b, a \in F \rightarrow b \in F \rightarrow (a \cap b) \in F)$
 $\wedge (\forall a b, a \subset b \rightarrow b \subset A \rightarrow a \in F \rightarrow b \in F)$.

² <https://github.com/1DGW/Formalization-of-Morse-Kelley-axiomatic-set-theory>.

Definition 3 (Ultrafilter). *Filter F over A is an ultrafilter if it satisfies:*

$$\forall a, a \subset A \implies a \in F \vee (A \sim a) \in F,$$

where $A \sim a$ represents the difference of A and a .

Ultrafilter has only one more condition than filter.

Definition ultraFilter F A := Filter F A
 $\wedge (\forall a, a \subset A \rightarrow a \in F \vee (A \sim a) \in F).$

Definition 4 (Principal Ultrafilter). *For every $a \in A$, the following set*

$$\{u : u \subset A \wedge a \in u\},$$

denoted as F_a , is an ultrafilter. Each F_a , corresponding to the element a of A , is called a principal ultrafilter over A .

Principal ultrafilters are a class of ultrafilters that can be concretely constructed. Formalizing this concept requires two steps: constructing the set F_a and verifying that F_a is indeed an ultrafilter.

Definition F A a := $\{ \lambda u, u \subset A \wedge a \in u \}$.

Property Fa_P1 : $\forall A a, F A a = \Phi \leftrightarrow a \notin A.$

Property Fa_P2_a : $\forall A a, \text{ultraFilter } (F A a) A \rightarrow a \in A.$

Property Fa_P2_b : $\forall A a, \text{Ensemble } A \rightarrow a \in A \rightarrow \text{ultraFilter } (F A a) A.$

The ultrafilters that are not principal ones are called non-principal ultrafilters. Besides, they are also known as free ultrafilters.

Definition 5 (Free Ultrafilter). *Ultrafilter F over A is a free ultrafilter if it satisfies:*

$$\forall a, a \text{ is a finite subset of } A \implies a \notin F.$$

Free ultrafilter has only one more condition than ultrafilter as well.

Definition free_ultraFilter F A := ultraFilter F A
 $\wedge (\forall a, a \subset A \rightarrow \text{Finite } a \rightarrow a \notin F).$

where “ $a \subset A$ ” and “ $\text{Finite } a$ ” represent “ a is a finite subset of A ”.

The following formalization indicates that each principal ultrafilter is not a free ultrafilter, and each non-free ultrafilter is a certain principal ultrafilter.

Theorem FT2_a : $\forall A a, \text{Ensemble } A \rightarrow a \in A$
 $\rightarrow \text{ultraFilter } (F A a) A \wedge \sim \text{free_ultraFilter } (F A a) A.$

Theorem FT2_b : $\forall F O A, \text{ultraFilter } F O A$
 $\rightarrow \sim \text{free_ultraFilter } F O A \rightarrow (\exists a, a \in A \wedge F O = F A a).$

Definition 6 (Fréchet Filter). *The Fréchet filter over set A is denoted as:*

$$F_\sigma = \{a : a \subset A \wedge A \sim a \text{ is finite}\}.$$

Definition Fσ A := $\{ \lambda a, a \subset A \wedge \text{Finite } (A \sim a) \}$.

Property Fσ_is_just_Filter : $\forall A, \sim \text{Finite } A \rightarrow \text{Ensemble } A$
 $\rightarrow \text{Filter } (F_\sigma A) A \wedge \sim \text{ultraFilter } (F_\sigma A) A$
 $\wedge (\forall a, a \subset A \rightarrow \text{Finite } a \rightarrow a \notin (F_\sigma A)).$

“`Fσ_is_just_Filter`” verifies that for each infinite set A , $F_σ$ (over A) is merely a filter but not an ultrafilter over A ; meanwhile, every finite subset of A is not in $F_σ$. Nevertheless, $F_σ$ is a special filter, because free ultrafilters are equivalent to the ultrafilters that contain $F_σ$, although $F_σ$ itself is merely a filter. This suggests the importance of $F_σ$ in the quest for non-principal ultrafilters.

Proposition `Fσ_and_free_ultrafilter` : $\forall F A, \text{Ensemble } A$
 $\rightarrow \sim \text{Finite } A \rightarrow \text{ultraFilter } F A$
 $\rightarrow \text{free_ultraFilter } F A \leftrightarrow (F\sigma A) \subset F.$

3.2 Extension of Filters

The process of constructing NPAUF involves iteratively extending specific filters or filter bases, in which an important theorem – Filter Extension Principle (FEP), also known as “Ultrafilter Theorem” [25], a consequence of the Axiom of Choice – will be utilized.

To describe FEP, the definition of the finite intersection property is required.

Definition 7 (Finite Intersection Property). G possesses the finite intersection property if the intersection of a finite number of elements in G is non-empty:

$$\forall a_1, a_2, \dots, a_n \in G, a_1 \cap a_2 \cap \dots \cap a_n \neq \emptyset.$$

Definition `Finite_Intersection` $G := \forall A, A \subset G \rightarrow \text{Finite } A \rightarrow \cap A \neq \emptyset.$

In code, “ $\cap A$ ” represents the intersection of all elements in A (i.e., $a_1 \cap a_2 \cap \dots \cap a_n$), which is not an empty set (“ $\cap A \neq \emptyset$ ”).

Theorem (FEP). For each set A , if the subset family G of A (i.e., $G \subset 2^A$) possesses finite intersection property, then there exists

- 1) a filter base B over A satisfying $G \subset B$;
- 2) a filter F over A satisfying $G \subset F$;
- 3) an ultrafilter F over A satisfying $G \subset F$.

The filter base and filter extended from G are constructive. For every subset family G of A , if G possesses finite intersection property, it can be extended to form a filter base B as follows:

$$B = \{a_1 \cap a_2 \cap \dots \cap a_n : a_1, a_2, \dots, a_n \in G (n \geq 1)\};$$

and extended to form a filter F as:

$$F = \{u : u \subset A \wedge \exists a_1, a_2, \dots, a_n \in G, a_1 \cap a_2 \cap \dots \cap a_n \subset u (n \geq 1)\}.$$

Definition `FilterBase_from` $G :=$
 $\{\lambda u, \exists S, S \subset G \wedge \text{Finite } S \wedge u = \cap S \setminus\}.$

Definition `Filter_from` $G A :=$
 $\{\lambda u, u \subset A \wedge \exists S, S \subset G \wedge \text{Finite } S \wedge \cap S \subset u \setminus\}.$

Notation “ $\langle G \rangle^b$ ” := `(FilterBase_from G) : filter_scope.`

Notation “ $\langle G \mid A \rangle^f$ ” := `(Filter_from G A) : filter_scope.`

In code, it is self-evident that the set represented by “S” is not empty. Because in MK system, $\cap\emptyset$ is a proper class that cannot belong to any other classes. This guarantees the condition $n \geq 1$.

The following formalization verifies that the extension in the manner described above does result in filter bases and filters that contain G respectively.

```

Lemma Filter_Extension1 :  $\forall G A, G \langle \rangle \Phi \rightarrow G \subset \text{pow}(A)$ 
  -> Finite_Intersection G ->  $G \subset (\langle G \rangle \rightarrow^b) \wedge \text{FilterBase } (\langle G \rangle \rightarrow^b) A.$ 
Lemma Filter_Extension_1_and_2 :  $\forall G A, G \langle \rangle \Phi \rightarrow G \subset \text{pow}(A) \rightarrow \text{Ensemble } A$ 
  -> Finite_Intersection G ->  $G \subset (\langle G|A \rangle \rightarrow^f) \wedge \text{Filter } (\langle G|A \rangle \rightarrow^f) A.$ 

```

As for the extension from G to ultrafilters, the proof requires the use of the Axiom of Choice [30], detailed in [10]. The formalization is listed here:

```

Theorem Filter_Extension_Principle :  $\forall G A, G \langle \rangle \Phi \rightarrow G \subset \text{pow}(A) \rightarrow \text{Ensemble } A$ 
  -> Finite_Intersection G ->  $\exists F, G \subset F \wedge \text{ultraFilter } F A.$ 

```

Given that filters possess the finite intersection property (straightforward to prove), FEP also asserts that every filter can be extended to an ultrafilter. Since non-principal ultrafilters are those ultrafilters containing F_σ (which is illustrated in the formalization of Definition 6), non-principal ultrafilters can be obtained by directly extending F_σ with FEP.

3.3 Arithmetical Ultrafilter

Just as its name implies, NPAUF is an arithmetical ultrafilter (AUF). So the definition of AUF should be introduced, before which we present the formalization of some involved concepts.

Definition 8 (Image Set). *Let f be a function and A be a subset of domain of f , then the image set of f at A is denoted as:*

$$f[A] = \{u : \exists x, u = f(x) \wedge x \in A\}.$$

Definition 9 (Preimage Set). *Let f be a function and A be a subset of range of f , then the preimage set of f at A is denoted as:*

$$f^{-1}[A] = \{u : u \in \text{domain } f \wedge f(u) \in A\}.$$

```

Definition ImageSet f A :=  $\setminus\{ \lambda u, \exists m, u = f[m] \wedge m \in A \setminus \}.$ 

```

```

Notation "f [ A ]" := (ImageSet f A)(at level 5) : filter_scope.

```

```

Definition PreimageSet f A :=  $\setminus\{ \lambda u, u \in \text{dom}(f) \wedge f[u] \in A \setminus \}.$ 

```

```

Notation "f ^{-1}[ A ]" := (PreimageSet f A)(at level 5) : filter_scope.

```

The image set at A is the set of values of f at members in A . In the formalization, A is not restricted to a subset of the domain of f , making its application scope broader.

Definition 10 (Ultrafilter Transformation). *Let F be an ultrafilter, and f be a function in B^A , then the following set*

$$f\langle F \rangle = \{u : u \subset B \wedge f^{-1}[u] \in F\}$$

can be proven an ultrafilter, where B^A represents the class consisting of all functions whose domain is A and range contained in B . $f\langle F \rangle$ is called a transformation of ultrafilter F under function f .

```

Definition Transform F f B := \{ \lambda u, u \subset B /\ f^{-1}[u] \in F \}.
Notation "f \langle F | B \rangle" := (Transform F f B)(at level 5) : filter_scope.

Definition \beta A := \{ \lambda u, ultraFilter u A \}.

```

```

Theorem FT4 : \forall F f A B, F \in (\beta A) -> Function f
-> dom(f) = A -> ran(f) \subset B -> Ensemble B -> f\langle F|B \rangle \in (\beta B).

```

The formalization “Transform” actually requires three parameters, therefore the formal notation looks a little more complicated than that in mathematics. “ βA ” represents the ultrafilter space that consists of all ultrafilters over A . “FT4” is the formal theorem that the ultrafilter F over A can be transformed to be the ultrafilter $f\langle F \rangle$ over B by the function f .

Definition 11 (F-Equivalence). Let F be an ultrafilter over A , then two functions f and g in B^A are F -equivalent if and only if

$$\{u : u \in A \wedge f(u) = g(u)\} \in F,$$

namely f and g are equal on a set in F , which is denoted as $f =_F g$.

```

Definition AlmostEqual f g A B F := Function f /\ Function g
/\ dom(f) = A /\ dom(g) = A /\ ran(f) \subset B /\ ran(g) \subset B
/\ F \in (\beta A) /\ \{ \lambda u, u \in A /\ f[u] = g[u] \} \in F.

```

Definition 12 (Arithmetical Ultrafilter). For every ultrafilter F over an infinite set A , F is an arithmetical ultrafilter if and only if:

$$\forall f, g \in A^A, f\langle F \rangle = g\langle F \rangle \implies f =_F g.$$

```

Definition Arithmetical_ultraFilter F A := \sim Finite A /\ F \in (\beta A)
/\ (\forall f g, Function f -> Function g
-> dom(f) = A -> dom(g) = A -> ran(f) \subset A -> ran(g) \subset A
-> f\langle F|A \rangle = g\langle F|A \rangle -> AlmostEqual f g A A F).

```

The formal definition consists of three assumptions: “ \sim Finite A ” indicates that A is an infinite set, “ $F \in (\beta A)$ ” represents that F is an ultrafilter over A , and the last assumption rules that for each function f and g , $f\langle F \rangle = g\langle F \rangle$ implies that f and g are F -equivalent.

It can be verified that every principal ultrafilter is an AUF:

```

Theorem FT9 : \forall A a, Ensemble A -> \sim Finite A -> a \in A
-> Arithmetical_ultraFilter (F A a) A.

```

The AUFs that are not principal ultrafilters are called non-principal arithmetical ultrafilters (NPAUFs).

4 Existence of Non-Principal Arithmetical Ultrafilter over ω

With the use of the Continuum Hypothesis (CH), NPAUF over ω can be constructed [29,30]. ω is the notation utilized to represent the set of natural numbers in set theory. The concepts and notations introduced in the last section should be instantiated to ω , which is implemented by the command “**Notation**”.

```

Notation F $\sigma$  := (filter.F $\sigma$   $\omega$ ).
Notation F := (filter.F  $\omega$ ).
Notation  $\beta_\omega$  := (arithmetical_ultrafilter. $\beta$   $\omega$ ).
Notation "f  $\langle$  F0  $\rangle$ " := (f(F0| $\omega$ ))(at level 5).
Notation "f =_ F0 g" := (arithmetical_ultrafilter.AlmostEqual f g  $\omega$   $\omega$  F0)
(at level 10, F0 at level 9).
Notation " $\langle$  G  $\rangle$  $\rightarrow^f$ " := ( $\langle$ G| $\omega$  $\rangle$  $\rightarrow^f$ ).
    
```

Then the meanings of the above notations, in sequence, are: Fréchet filter over ω , principal ultrafilters over ω , the ultrafilter space over ω , the transformation of ultrafilter $F0$ under the function $f(\in \omega^\omega)$, functions $f, g (\in \omega^\omega)$ are $F0$ -equivalent, the filter extended from G .

4.1 Lemmas

This subsection introduces the formalization of the main lemmas (not all of them) used in the proof process presented in Sect. 4.3. Some other lemmas, not reflected in the macro-level proof framework, are not elaborated.

Lemma 1. *Assume that B is a countable filter base and contains the Fréchet filter ($F_\sigma \subset B$), F is the filter extended from B using the Filter Extension Principle, f and g are functions in $\omega^\omega (= \{u : u \text{ is a function} \wedge \text{domain } u = \omega \wedge \text{range } u \subset \omega\})$. If*

$$\{u : u \in \omega \wedge f(u) \neq g(u)\} \in F,$$

and each finite set A contained in ω ($A \subset \omega$) satisfies

$$f^{-1}[g[A]] \cup g^{-1}[f[A]] \notin F,$$

then there exists a subset $a \subset \omega$ such that $B \cup \{a\}$ has the finite intersection property and $f[a] \cap g[a] = \emptyset$.

A countable set B is usually introduced as “ B and ω are equipotent” or “ B is finite” [30]. The former means B and ω have the same cardinality, which is demonstrated as “ $P(B) = P(\omega) = \omega$ ”³ in MK; while the latter is “ $P(B) \in \omega$ ”. P is the function that maps a set to its cardinality, and the cardinality of B is formalized as “ $P[B]$ ” in Coq [18,24,35]. Thus we consider “ $P[B] \in \omega + P[B] = \omega$ ” in Coq suitable for describing countability.

³ Note that ω has many meanings in set theory. It can represent the set of all natural numbers, meanwhile it is both the first infinite ordinal number and the first infinite cardinal number. [18].

```

Lemma Existence_of_NPAUF_Lemma_a : ∀ B f g, P[B] ∈ ω \ / P[B] = ω
-> FilterBase B ω -> Fσ ⊂ B -> Function f -> Function g
-> dom(f) = ω -> dom(g) = ω -> ran(f) ⊂ ω -> ran(g) ⊂ ω
-> \{ λ u, u ∈ ω / \ f[u] <> g[u] \} ∈ ((B)→f)
-> (∀ A, A ⊂ ω -> Finite A -> (f-1[g[A]] ∪ g-1[f[A]]) ∉ ((B)→f))
-> (∃ a, a ⊂ ω / \ Finite_Intersection (B ∪ [a]) / \ f[a] ∩ g[a] = ∅).
    
```

where “ $\langle B \rangle \rightarrow^f$ ” represents the filter extended from filter base B .

Lemma 2. $\omega^\omega \times \omega^\omega$ is equipotent to $2^\omega (= \{u : u \subset \omega\})$.

The symbol 2^ω represents the power set of ω , formalized as “pow(ω)”. Symbol “ \times ” is the Cartesian product operation, and the same notation is adopted in the MK formalization.

```

Definition Cartesian x y := \{ \ λ u v, u ∈ x / \ v ∈ y \}.
Notation "x × y" := (Cartesian x y) (at level 2, right associativity).
    
```

```

Lemma Existence_of_NPAUF_Lemma_b :
\{ λ f, Function f / \ dom(f) = ω / \ ran(f) ⊂ ω \}
× \{ λ f, Function f / \ dom(f) = ω / \ ran(f) ⊂ ω \} ≈ pow(ω).
    
```

Lemma 3. If a non-empty set A is countable and the cardinality of all of its elements is ω , then the cardinality of $\bigcup A$ (the union of all elements of A) is ω .

The formal proof of this lemma can be split into two cases: one where the cardinality of A is exactly ω , and the other where A is a non-empty finite set.

```

Lemma Existence_of_NPAUF_Lemma_c1 : ∀ A, P[A] = ω
-> (∀ a, a ∈ A -> P[a] = ω) -> P[∪A] = ω.
Lemma Existence_of_NPAUF_Lemma_c2 : ∀ A, Finite A -> A <> ∅
-> (∀ a, a ∈ A -> P[a] = ω) -> P[∪A] = ω.
    
```

Lemma 4. The cardinality of the Fréchet filter (F_σ) over ω is ω .

Lemma 5. If G is equipotent to ω , then the filter base extended from G using the Filter Extension Principle is equipotent to ω as well.

Lemma 4 and Lemma 5 are straightforward to be formalized as listed here:

```

Lemma Existence_of_NPAUF_Lemma_d : P[Fσ] = ω.
Lemma Existence_of_NPAUF_Lemma_e : ∀ G, P[G] = ω -> P[\langle G \rangle →b] = ω.
    
```

4.2 Formalization of the Continuum Hypothesis

The Continuum Hypothesis (CH) was proposed by Cantor, who, after extensive research, believed that any uncountable subset of the real number set \mathbf{R} is equipotent to \mathbf{R} itself [30].

According to the known results, the cardinality of an uncountable set is greater than ω , while the cardinality of \mathbf{R} is equal to the cardinality of the powerset of ω (i.e., $P(\mathbf{R}) = P(2^\omega)$) [30]. At the same time, $P(2^\omega)$ is strictly greater than ω [18]. Therefore, CH actually asserts that there are no cardinal numbers between ω and $P(2^\omega)$.

Gödel and Cohen have proven that CH is consistent and independent of general set theory [4,7,11,12], which allows people to confidently use CH in certain specific situations, such as in proving the existence of NPAUF.

Given that CH cannot be proven in the MK system, we choose the command “**Axiom**” to formalize it. The statements declared by **Axiom** do not require verification and can be directly invoked.

```
Axiom CH : ∀ c, c ∈ C -> ω < c -> P[pow(ω)] ≤ c.
```

where “**C**” is the formal notation of the class consisting of all cardinal numbers (see Appendix A), and the notation “**<**” and “**≤**” are used to compare the sizes of two cardinal numbers.

```
Definition Less x y := x < y.
```

```
Notation "x < y" := (Less x y) (at level 67, left associativity).
```

```
Definition LessEqual x y := x < y ∨ x = y.
```

```
Notation "x ≤ y" := (LessEqual x y) (at level 67, left associativity).
```

Note that in the context of set theory and cardinalities, $x \in y$ exactly means that cardinal number x is less than cardinal number y (i.e., $x < y$).

This formalization essentially acknowledges: for each cardinal number c , if c is strictly greater than ω , then c must be greater than or equal to $P(2^\omega)$, meaning there are no cardinal numbers between ω and $P(2^\omega)$ (strictly greater than ω and strictly less than $P(2^\omega)$). This accurately reflects the mathematical meaning of the Continuum Hypothesis.

4.3 The Existence of Non-principal Arithmetical Ultrafilters

The formal description that there exists NPAUF is straightforward:

```
Theorem Existence_of_NPAUF : ∃ F0, Arithmetical_ultraFilter F0 ω
  ∧ (∀ n, F0 <> F n).
```

where “**F0 <> F n**” indicates that the specific arithmetical ultrafilter $F0$ is not any principal ultrafilter. The difficulty lies in verifying the tedious proof process. The manual proof is introduced by Wang in [30]. The overall proof strategy is divided into two parts.

First, to construct a specific infinite sequence of filter bases:

$$B_0 \subset B_1 \subset B_2 \subset \dots \subset B_\alpha \subset \dots (\alpha < P(2^\omega))$$

where $\alpha < P(2^\omega)$ represents that the ordinal number α is less than $P(2^\omega)$ (i.e., $\alpha \in P(2^\omega)$). Such a sequence with mutual containment relationships is called a “nest” in MK [18,24,35].

Second, to extend $\bigcup\{B_\alpha : \alpha < P(\omega)\}$ (the union of all B_α) into an ultrafilter p using FEP. Then p can be proven exactly an NPAUF.

Thus the most important part of the proof is the construction of the nest of filter bases above. According to Lemma 2, every ordered pair (f_α, g_α) belonging to ω^ω can be enumerated using the ordinal numbers less than 2^ω :

$$(f_0, g_0), (f_1, g_1), (f_2, g_2), \dots, (f_\alpha, g_\alpha), \dots (\alpha < P(2^\omega))$$

Then let $e_\alpha = \{n : n \in \omega \wedge f_\alpha(n) = g_\alpha(n)\}$. The idea of constructing the filter base nest is to continually incorporate e_α into F_σ and extend the resulting sets into filter bases, until all e_α have been exhausted.

We present the entire construction process in a procedural manner in Fig. 1, which serves as the guideline of the formalization.

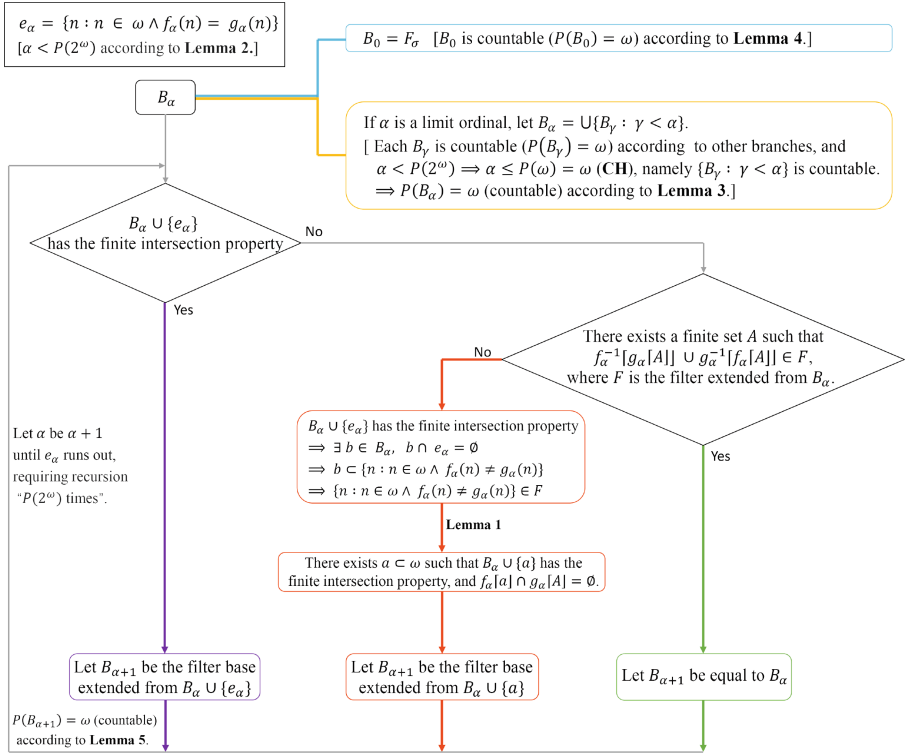


Fig. 1. the construction process of the filter base nest

It can be observed from Fig. 1 that the construction is divided into five cases, distinguished by different colors. When α is 0, it corresponds to the blue branch ($B_0 = F_\sigma$); when α is a limit ordinal, it corresponds to the yellow branch (where CH is needed). A limit ordinal cannot be the successor of any ordinals. For example, ω is a limit ordinal, because there exists no ordinal n such that $\omega = n + 1$. When B_α is known, the construction of $B_{\alpha+1}$ is then divided into 3 cases respectively corresponding to the purple, red and green branches.

Since the looping process in Fig. 1 will only stop when e_α is exhausted, and the number of e_α is infinite, this construction process is actually a transfinite recursion process. We need to introduce the transfinite recursion theorem, which has been introduced as the 128th theorem in the MK system [18, 24, 35].

Theorem128 (Transfinite Recursion). *For each h there exists a function r such that domain of r is an ordinal and $r(x) = h(r|x)$ holds for each ordinal number x .*

Theorem MKT128a : $\forall h, \exists r, \text{Function } r \wedge \text{Ordinal } \text{dom}(r) \wedge (\forall x, \text{Ordinal_Number } x \rightarrow r[x] = h[r|(x)])$.

where the notation $r|x$ represents the restriction of r to x , and it is formalized as “ $r|(x)$ ” in MK formalization. Transfinite recursion indicates that the value of r at ordinal number x can be determined by the values at all ordinal numbers less than x , with the determining rule given by h . Therefore, the key to constructing a transfinite recursive function lies in constructing its corresponding h .

Additionally, in the red branch where Lemma 1 needs to be used, a subset a of ω is needed. However, the proof of Lemma 1 is non-constructive [30], so we need the Axiom of Choice to perform the operation of “selecting a ”. The Axiom of Choice is formalized as follows in MK system:

Definition ChoiceFunction $c :=$
 $\text{Function } c \wedge (\forall x, x \in \text{dom}(c) \rightarrow c[x] \in x)$.
Axiom AxiomIX : $\exists c, \text{ChoiceFunction } c \wedge \text{dom}(c) = \mu \sim [\Phi]$.

A choice function c can obtain a specific member from a set in its domain (i.e., $c(x) \in x$ where $x \in \text{domain } c$), which is defined by the formal definition “ChoiceFunction c ”. And “AxiomIX” acknowledges that there exists such a choice function c whose domain is the universe (i.e., the class consisting of all sets), which means that for each set x , c can select the element $c(x)$ from it.

With the above analysis, we can now implement the construction of B_α .

Firstly, the critical set e_α needs to be formalized. Using Lemma 2, we can obtain a bijective function between $P(2^\omega)$ and $\omega^\omega \times \omega^\omega$. We denote this bijective function as φ in the Coq proof environment:

```
...
φ : Class
H1 : Function φ
H4 : Function φ-1
H5 : dom(φ) = P[pow(ω)]
H6 : ran(φ) = { λ f, Function f ∧ dom(f) = ω ∧ ran(f) ⊂ P[pow(ω)] \ }
      × { λ f, Function f ∧ dom(f) = ω ∧ ran(f) ⊂ P[pow(ω)] \ }
      -----(1/1)
∃ F0, Arithmetical_ultraFilter F0 ω ∧ (∀ n, F0 <> F n)
```

where “H_” can be regarded as the labels of the conditions introduced by default during the proof. e_α should be formalized as a function whose domain is $P(2^\omega)$ and value at α is $\{n : n \in \omega \wedge f_\alpha(n) = g_\alpha(n)\}$, where $f_\alpha(n)$ and $g_\alpha(n)$ need to be represented by φ . We utilize the tactic “set” to define this function:

```
set (e := { λ α v, α ∈ P[pow(ω)] ∧ v = { λ n, n ∈ ω
      ∧ (First φ[α])[n] = (Second φ[α])[n] \} \})
```

The tactic “set” is to introduce a specific definition, which is similar to the command “Definition”. But a term defined by “set” only works in the proof environment it lives in.

According to the setting of φ , $\varphi[\alpha]$ represents the ordered pair (f_α, g_α) . “First” and “Second”, from MK system, can respectively take the first and second coordinates of ordered pairs. Thus for each $\alpha \in P(2^\omega)$, $\{n : n \in \omega \wedge f_\alpha(n) =$

$g_\alpha(n)\{= e_\alpha\}$ is formalized as “**e**[α]”, which is verified and added to the premises in proof environment by the tactic “**assert**”.

```
assert (Function e /\ dom(e) = P[pow(omega)]) as [].
assert (forall alpha, alpha in P[pow(omega)] -> e[alpha] = { lambda m, m in omega
  /\ (First phi[alpha])[m] = (Second phi[alpha])[m] })
```

The proposition asserted by the tactic “**assert**” needs to be proven, after which it can be invoked as a condition in the proof environment.

```
...
e := { lambda alpha v, alpha in P[pow(omega)]
  /\ v = { lambda n, n in omega /\ (First phi[alpha])[n] = (Second phi[alpha])[n] } }
H9a : Function e
H9b : dom(e) = P[pow(omega)]
H9 : forall alpha, alpha in P[pow(omega)]
  -> e[alpha] = { lambda m, m in omega /\ (First phi[alpha])[m] = (Second phi[alpha])[m] }
----- (1/1)
exists F0, Arithmetical_ultraFilter F0 omega /\ (forall n, F0 <> F n)
```

Then we use the tactic “**destruct**” to break down “AxiomIX” (the Axiom of Choice), and add the conditions obtained to the proof environment.

```
destruct AxiomIX as [c[]].
...
c : Class
H10 : ChoiceFunction c
H11 : dom(c) = mu ~ [Phi]
----- (1/1)
exists F0, Arithmetical_ultraFilter F0 omega /\ (forall n, F0 <> F n)
```

If there is a parameter x representing x , $c[x]$ represents a specific element of x .

With the use of choice function, we construct the h required for transfinite recursion with the tactic “**set**”. The formal description h is divided into 5 cases by the “ \setminus ” symbol, each corresponding to one of the 5 branches in Fig. 1.

```
set (h := { lambda lambda u v, Ordinal dom(u) /\
  ((dom(u) = Phi /\ v = Fsigma)
  /\ (dom(u) <> Phi /\ ((exists m, LastMember m E dom(u)
  /\ ((Finite_Intersection (u[m] union e[m])) /\ v = (u[m] union e[m])) -> b)
  /\ (~ Finite_Intersection (u[m] union e[m]))
  /\ ((exists b, Finite b /\ b subset omega /\ ((First phi[m])^-1[(Second phi[m])[b]]
  union (Second phi[m])^-1[(First phi[m])[b]]) in (u[m]) -> f /\ v = u[m])
  /\ ((forall b, Finite b -> b subset omega -> ((First phi[m])^-1[(Second phi[m])[b]]
  union (Second phi[m])^-1[(First phi[m])[b]]) not in (u[m]) -> f)
  /\ v = (u[m] union [c[{ lambda lambda w, w subset omega /\ Finite_Intersection (u[m] union [w])
  /\ (First phi[m])[w] in (Second phi[m])[w] = Phi}]) -> b))))))
  /\ (~ (exists m, LastMember m E dom(u)) /\ v = union (ran(u)))) } } \setminus \setminus
```

Here, “**LastMember**” (derived from MK formalization) is a formalization not introduced in previous sections. It is used to indicate that an element is the last member of another class under a specific relation. For example, “**LastMember** $x \in y$ ” indicates that x is the last member of y under the “ \in ” relation (E represents the “ \in ” relation in MK formalization, see Appendix A). So the use of “**LastMember**” here is to determine whether there exists a last element in α under the “ \in ” relation, that is, whether α is a limit ordinal.

Now we can apply the transfinite recursion theorem (MKT128a) to h and add the recursive function r to the proof environment.

```
destruct (MKT128a h) as [r[?[]]].
...
r : Class
H13 : Function r
H14 : Ordinal dom(r)
H15 : ∀ x, Ordinal_Number x -> r[x] = h[r|(x)]
----- (1/1)
∃ F0, Arithmetical_ultraFilter F0 ω /\ (∀ n, F0 <> F n)
```

The range of $r|P(2^\omega)$ (restriction of r to $P(2^\omega)$) is exactly the filter base nest we want because of the following assertions that can be verified:

```
assert (r[Φ] = Fσ).
assert (∀ m n, m ∈ n -> r[m] ⊂ r[n]).
assert (∀ n, n ∈ dom(r) -> FilterBase r[n] ω).
assert (∀ n, n ∈ dom(r) -> n ∈ P[pow(ω)] -> P[r[n]] = ω).
assert (P[pow(ω)] ⊂ dom(r)).
assert (Function (r|(P[pow(ω)]))).
assert (dom(r|(P[pow(ω)])) = P[pow(ω)]).
```

From the first four assertions, it can be observed that the value of r satisfies:

$$r(0)(= F_\sigma) \subset r(1) \subset r(2) \subset \dots \subset r(n) \subset \dots (n \in \text{domain of } r),$$

and each $r(n)$ is a countable (CH is needed) filter base over ω . From the last three assertions, $P(2^\omega)$ is contained in domain of r , thus the range of $r|P(2^\omega)$ is the target filter base nest.

Finally, according to the manual proof, we extend $\bigcup(\text{range } r|P(2^\omega))$ to an ultrafilter p using FEP, and p is a non-principal ultrafilter.

```
...
p : Class
H24 : ( ∪ran(r|(P[pow(ω)])) ) →f ⊂ p
H25 : ultraFilter p ω
H26 : ∀ n, p <> F n
----- (1/1)
∃ F0, Arithmetical_ultraFilter F0 ω /\ (∀ n, F0 <> F n)
```

As for the verification that p is an arithmetical ultrafilter, it is relatively mechanical, which only requires proving each of the 5 cases in h (corresponding to the 5 branches in Fig. 1) one by one.

5 Conclusion

This work is grounded in MK formalization [18, 24, 35], and is the foundation that paves the way for the formal verification of the real number theory introduced in [27, 30]. The contributions are the Coq formalization and verification of proofs regarding filter, ultrafilter, arithmetical ultrafilter, CH and most importantly, non-principal arithmetical ultrafilter.

The work in this paper comprises 6 (.v)files and approximately 5,500 lines of Coq code (excluding MK formalization). All code has been successfully executed in the Coq IDE. Figure 2 shows the dependency of our Coq development (.v)files.

The MK part (red boxes) is the previous work [24, 35] this paper based on, and the part of green boxes is our development in this paper. In `operations_on_ω`

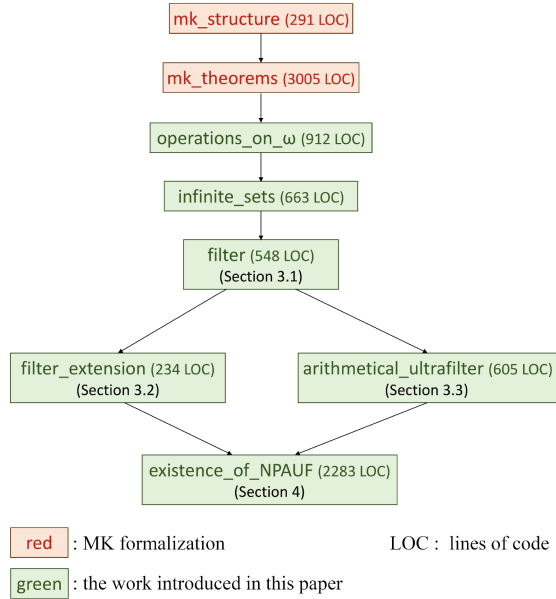


Fig. 2. The dependency graph of our Coq development (.v)files

and `infinite_sets`, we supplemented the formalization of basic operations on ω and properties of infinite sets, which are not explicitly mentioned in this paper but are necessary for the proof that F_σ is a filter but not an ultrafilter. Readers can verify the details in our Coq code.

Formalization indeed allows for rigorous verification of mathematical proofs. As introduced in Sect. 4.3, a recursive function is essential to prove the existence of NPAUF; and obtaining the set a derived from Lemma 1 relies on the Axiom of Choice. These are all details not explicitly reflected in the manual proof description in [30]. Certainly, we cannot deny the correctness of manual proofs for this, but formalization undoubtedly strengthens the rigor. As proposed by Avigad and Harrison, “with the help of computational proof assistants, formal verification could become the new standard for rigor in mathematics.” [1]

To advance further, the formalization work of the real number theory (based on NPAUF) proposed by Wang in [27, 30] is currently in progress. Afterward, as mentioned in the Introduction, we plan to use NPAUF for the construction of hyper-real numbers, and establish a formalized system of non-standard analysis.

Acknowledgments. Our work is funded by National Natural Science Foundation (NNSF) of China under Grant 61936008.

A Partial Definitions and Notations of MK

Mathematical Meaning	Mathematical Symbol
Definition in Coq ⁴	Notation in Coq
complement of x Complement $x := \setminus \{ \lambda y, y \notin x \}$	$\neg x$ $\neg x$
difference of x and y ; complement of y relative to x Setminus $x \ y := x \cap (\neg y)$	$x \sim y$ $x \sim y$
void class; empty set $\Phi := \setminus \{ \lambda x, x \ltimes x \}$	$\emptyset; 0$ Φ
universe, the class including all sets $\mu := \setminus \{ \lambda x, x = x \}$	\mathcal{U} μ
class of the intersection of the members of x Element_I $x := \setminus \{ \lambda z, \forall y, y \in x \rightarrow z \in y \}$	$\bigcap x$ $\cap x$
class of the union of the members of x Element_U $x := \setminus \{ \lambda z, \exists y, z \in y \wedge y \in x \}$	$\bigcup x$ $\cup x$
x is a subclass of y ; x is contained in y Included $x \ y := \forall z, z \in x \rightarrow z \in y$	$x \subset y; x \subseteq y$ $x \subset y$
power class of x PowerClass $x := \setminus \{ \lambda y, y \subset x \}$	2^x pow (x)
singleton class of x Singleton $x := \setminus \{ \lambda z, x \in \mu \rightarrow z = x \}$	$\{x\}$ $[x]$
unordered pair of x and y Unordered $x \ y := [x] \cup [y]$	$\{xy\}$ $[x y]$
ordered pair of x and y Ordered $x \ y := [[x] [x y]]$	(x, y) $[x, y]$
the first coordinate of z First $z := \cap \cap z$	$1^{st} \text{ coord } z$ -
the second coordinate of z Second $z := (\cap \cup z) \cup (\cup \cup z) \sim (\cup \cap z)$	$2^{st} \text{ coord } z$ -
r is a relation iff its members are ordered pairs Relation $r := \forall z, z \in r \rightarrow \exists x \ y, z = [x, y]$	- -
relation inverse to r Inverse $r := \setminus \{ \lambda x \ y, [y, x] \in r \}$	r^{-1} r^{-1}
f is a function Function $f := \text{Relation } f$ $\wedge (\forall x \ y \ z, [x, y] \in f \rightarrow [x, z] \in f \rightarrow y = z)$	- -
domain of the class f Domain $f := \setminus \{ \lambda x, \exists y, [x, y] \in f \}$	domain f $\text{dom}(f)$
range of the class f Range $f := \setminus \{ \lambda y, \exists x, [x, y] \in f \}$	range f $\text{ran}(f)$
value of f at x or image of x under f Value $f \ x := \cap (\setminus \{ \lambda y, [x, y] \in f \})$	$f(x)$ $f[x]$
f is a 1-1 function (bijective function) Function1_1 $f := \text{Function } f \wedge \text{Function } (f^{-1})$	- -
class consisting of functions whose domain is x and range is contained in y Exponent $y \ x :=$ $\setminus \{ \lambda f, \text{Function } f \wedge \text{dom}(f) = x \wedge \text{ran}(f) \subset y \}$	y^x -
cartesian product of x and y	$x \times y$

⁴ All definitions are defined with the command “**Definition**”.

Cartesian $x \times y := \{\lambda \ u \ v, u \in x \wedge v \in y\}$	$x \times y$
restriction of f to x	$f _x$
Restriction $f \upharpoonright x := f \cap (x \times \mu)$	$f \upharpoonright (x)$
x is r -related to y or x r -precedes y	xry
Relation $x \ r \ y := [x,y] \in r$	-
r connects x (trichotomy)	-
Connect $r \ x := \forall u \ v, u \in x \rightarrow v \in x$ $\rightarrow (Rrelation \ u \ r \ v) \vee (Rrelation \ v \ r \ u) \vee (u = v)$	-
x is full (each member of a member of x is a member of x)	-
Full $x := \forall m, m \in x \rightarrow m \subset x$	-
E is the \in -relation	E
E $:= \{\lambda \ x \ y, x \in y\}$	E
x is an ordinal	-
Ordinal $x := Connect \ E \ x \wedge Full \ x$	-
class consisting of all ordinal numbers ⁵	R
R $:= \{\lambda \ x, Ordinal \ x\}$	R
x is an ordinal number if and only if $x \in R$	-
Ordinal_Number $x := x \in R$	-
successor of x	$x'; x + 1$
PlusOne $x := x \cup [x]$	-
there exists a 1-1 function between x and y ;	$x \approx y$
x is equivalent (equipotent) to y ; x and y are equipollent	
Equivalent $x \ y :=$ $\exists f, Function1_1 \ f \wedge dom(f) = x \wedge ran(f) = y$	$x \approx y$
x is a cardinal number	-
Cardinal_Number $x := Ordinal_Number \ x$ $\wedge (\forall y, y \in R \rightarrow y \in x \rightarrow \sim (x \approx y))$	-
class consisting of all cardinal numbers	C
C $:= \{\lambda \ x, Cardinal_Number \ x\}$	C
cardinality function that maps a set to its cardinality	P
P $:= \{\lambda \ x \ y, x \approx y \wedge y \in C\}$	P
set of (non-negative) integers (i.e., set of natural numbers)	ω
ω $:= \{\lambda \ x, Integer \ x\}$	ω
x is finite	-
Finite $x := P[x] \in \omega$	-

References

1. Avigad, J., Harrison, J.: Formally verified mathematics. Commun. ACM **57**(4), 66–75 (2014). <https://doi.org/10.1145/2591012>
2. Barras, B.: Sets in Coq, Coq in Sets. J. Formalized Reason. **3**(1), 29–48 (2010). <https://doi.org/10.6092/issn.1972-5787/1695>
3. Bartoszynski, T., Shelah, S.: There may be no hausdorff ultrafilters (2003). <https://doi.org/10.48550/arXiv.math/0311064>
4. Bell, J.L.: Set Theory: Boolean-Valued Models and Independence Proofs (Oxford Logic Guides 47), 3rd edn. Clarendon Press, Oxford (2005)
5. Bernays, P., Fraenkel, A.A.: Axiomatic Set Theory. North Holland Publishing Company, Amsterdam (1958)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Interactive Constructions. Springer, Berlin (2004). <https://doi.org/10.1007/978-3-662-07964-5>



⁵ R is an ordinal but not an ordinal number.

7. Cohen, P.J.: Set Theory and the Continuum Hypothesis. W.A.Benjamin Inc., New York (1966)
8. Comfort, W.W., Negrepointis, S.: The Theory of Ultrafilters. Springer, Berlin (1974). <https://doi.org/10.1007/978-3-642-65780-1>
9. The Coq Development Team. The Coq Proof Assistant Reference Manual (Version 8.19.0) (2024). <https://coq.inria.fr/doc/V8.19.0/refman/>. Accessed 8 July 2024
10. Dou, G., Yu, W.: Formalization of the filter extension principle (FEP) in coq. In: Zhang, L., Yu, W., Wang, Q., Laili, Y., Liu, Y. (eds.) Intelligent Networked Things. CINT 2024. CCIS, vol. 2138, pp. 95-106. Springer, Singapore (2024). https://doi.org/10.1007/978-981-97-3951-6_10
11. Gödel, K.: Consistency-proof for the generalized continuum-hypothesis. Proc. Natl. Acad. Sci. **25**(4), 220–224 (1939). <https://doi.org/10.1073/pnas.25.4.220>
12. Gödel, K.: The Consistency of the Axiom of Choice and of the Generalized Continuum Hypothesis With the Axioms of Set Theory. Princeton University Press, Princeton (1940)
13. Gonthier, G.: Formal proof - the Four Color Theorem. Not. Am. Math. Soc. **55**(11), 1382–1393 (2008). <https://www.ams.org/notices/200811/tx081101382p.pdf>
14. Gonthier, G., et al.: A machine-checked proof of the Odd Order Theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Interactive Theorem Proving. ITP 2013. LNCS, vol. 7998, pp. 163-179. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_14. https://www.cs.unibo.it/~asperti/PAPERS/odd_order.pdf
15. Hewitt, E.: Rings of real-valued continuous functions. I. Trans. Am. Math. Soc. **64**(1), 45-99 (1948). <https://sci-hub.st/10.1090/s0002-9947-1948-0026239-9>
16. Huet, G., Kahn, G., Paulin-Mohring, C.: The Coq Proof Assistant: A Tutorial (Version 8.5). Technical Report 178, INRIA (2016). <https://coq.inria.fr/tutorial/>
17. Jin, R.L.: Nonstandard analysis and its applications. Scientia Sinica Mathematica **46**(4), 371–408 (2016). <https://doi.org/10.1360/N012015-00266>. (in Chinese)
18. Kelley, J.L.: General Topology. Springer, New York (1955). <https://doi.org/10.1007/978-1-4757-4032-5>
19. Mendelson, E.: Introduction to Mathematical Logic, 4th edn. Chapman and Hall, London (1997)
20. Monk, J.D.: Introduction to Set Theory. McGraw-Hill, New York (1969)
21. Pierce, B.C., de Amorim, A.A., Casinghino, C., et al.: Software Foundation. <http://softwarefoundations.cis.upenn.edu/>. Accessed 8 July 2024
22. Robinson, A.: Non-standard Analysis, revised edn. North Holland Publishing Company, Amsterdam (1974)
23. Rubin, J.E.: Set Theory for the Mathematician. Holden Day, San Francisco (1967)
24. Sun, T., Yu, W.: A formal system of axiomatic set theory in Coq. IEEE Access **8**, 21510–21523 (2020). <https://doi.org/10.1109/ACCESS.2020.2969486>
25. Thomas, J.J.: The Axiom of Choice. North-Holland Publishing Company, Amsterdam (1973)
26. Wan, X., Xu, K., Cao, Q.: Coq formalization of ZFC set theory for teaching scenarios. Int. J. Softw. Inf. **13**(3), 323–357 (2023). <https://www.ijsi.org/ijsi/article/pdf/303>
27. Wang, F.: On a special kind of points in stone-čech compactification $\beta\omega$. J. China Univ. Sci. Tech. **28**(5), 567–570 (1998)
28. Wang, F.: A result on arithmetical ultrafilters. J. China Univ. Sci. Tech. **30**(5), 517–522 (2000)
29. Wang, F.: Arithmetical Ultrafilters: End-Extensons of \mathbf{N} in $\beta\mathbf{N}$. University of Science and Technology of China Press, Hefei (2016). (in Chinese)

30. Wang, F.: *Mathematical Foundations*, revised edn. Higher Education Press, Beijing (2018). (in Chinese)
31. Wang, H.: On Zermelo's and Von Neumann's axioms for set theory. *Proc. Natl. Acad. Sci. U.S.A.* **35**(3), 150–155 (1949). <https://doi.org/10.1073/pnas.35.3.150>
32. Werner, B.: Sets in types, types in sets. In: Abadi, M., Ito, T. (eds.) *TACS 1997*. LNCS, vol. 1281, pp. 530–546. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0014566>
33. Herbelin, H., Palmkog, K., et al.: An encoding of Zermelo-Fraenkel Set Theory in Coq (1996). <https://github.com/coq-contribs/zfc>
34. Wiedijk, F.: Formal proof - getting started. *Not. Am. Math. Soc.* **55**(11), 1408–1414 (2008). <https://www.ams.org/notices/200811/tx081101408p.pdf>
35. Yu, W., Sun, T., Fu, Y.: *A Machine Proof System for Axiomatic Set Theory*. Science Press, Beijing (2020). (in Chinese)
36. Zhang, Q.P.: *Set-Theory: Coq encoding of ZFC and formalization of the textbook elements of set theory*. <https://github.com/choukh/Set-Theory>



Observability of Boolean Control Networks: New Definition and Verification Algorithm

Guisen Wu¹ , Zhiming Liu¹ , and Jun Pang² 

¹ School of Computer and Information Science, Southwest University, Chongqing, China

zhimingliu88@swu.edu.cn

² Department of Computer Science, University of Luxembourg, Esch-sur-Alzette, Luxembourg

jun.pang@uni.lu

Abstract. Boolean Control Networks (*BCNs*) are extensively employed for modelling biological systems, attracting considerable attention from biologists and systems scientists, in particular, on the control theory of *BCNs*. This paper begins by demonstrating an erroneous definition of a specific property known as single-experiment observability, which was intended to represent the solvability of the *BCNs*' single-experiment observation problem. Subsequently, we propose a novel form of observability to redefine this property. With this new definition, the determination of initial states for larger sets of *BCNs* can now be achieved through a single experiment. Furthermore, we present a verification algorithm designed for our definition, which exhibits lower computational complexity compared to the algorithms used for verifying the previous definition of *BCNs*' single-experiment observability.

Keywords: Observability · Specification · Verification

1 Introduction

In the 1960s, Nobel Prize laureates Jacob and Monod proposed “Any cell contains several regulatory genes that act as switches and can turn one another on and off” [10], leading to the development of Boolean networks (*BNs*) [11]. *BNs* model biological systems using binary variables to represent genes and Boolean functions to describe their regulatory relationships. Boolean control networks (*BCNs*) extend *BNs* by incorporating external regulations and distinguishing between input, state, and output nodes [9]. *BCNs* are used in analyzing signaling networks [12, 13], drug target discovery [2], and solving evasion problems [19]. It is worth mentioning that *BCNs*, as a special variant of finite state machines (FSMs), have the set of initial states being the same as the set of states. Research on the control theory of *BCNs* can be transferred to FSMs [24], and FSMs are

an important and commonly used tool for modeling system behavior in formal methods.

The study of control theory in *BCNs* has garnered significant interest, resulting in various proposed problems for real-life scenarios [1, 3–7, 14–16, 18, 28, 29]. Tackling these problems generally involves four steps:

1. The real-life problem is formalized as a *BCN*'s control-theoretic problem.
2. An algorithm is devised to solve the control-theoretic problem.
3. The problem's solvability is formalized as a control-theoretic property, specifying the conditions a *BCN* must meet for the solution algorithm to succeed.
4. A verification algorithm is designed to check if a given *BCN* satisfies the control-theoretic property.

As can be seen from the four steps above, ensuring correct control-theoretic properties is vital for solving problems in *BCNs*. However, we show in this work that the current definition of single-experiment observability of *BCNs* is incorrect and needs revision. Redefining this property can improve its understanding and applicability in *BCNs*.

The observation problem, crucial for the quantitative analysis and identification of *BCNs*, remains a relevant and active topic [3, 5, 7, 27, 28, 30, 31]. It involves determining the initial state of a *BCN* by manipulating its inputs and observing its outputs, given knowledge of its updating rules [27]. This problem is divided into three sub-problems: multiple-experiment, single-experiment, and arbitrary-experiment observation problems. These sub-problems explore scenarios where (1) the input can be controlled and the initial state reset, (2) the input can be controlled but the initial state cannot be reset, and (3) the input cannot be controlled. The significance of the observability issue is manifestly evident in practical applications, such as in the context of gene regulatory networks that are amenable to modeling via *BCNs* [11], where direct measurement of all internal states is typically unattainable. The principle of observability enables the estimation and prognosis of the system's state from observable outputs, which is of paramount importance for the formulation of diagnostic and therapeutic strategies. To address these, four types of observability have been proposed: Type-I, II, III, & IV observability [3, 5, 7, 28], each providing distinct approaches and insights for solving the observation problem under various constraints [21].

Type-I observability in *BCNs* means a *BCN* is observable if, for any initial state, there exists an input sequence that can uniquely identify it from all other initial states [3]. Type-II observability, however, requires that for any two distinct initial states, there exists an input sequence to distinguish between them [28]. Initially, Type-I observability was proposed for scenarios where the *BCN*'s input can be controlled and the initial state can be reset. However, it was later replaced by Type-II observability because it is easier to satisfy [27]. In scenarios where the input can be controlled and the initial state can be reset, multiple experiments can be conducted to determine the initial state, hence Type-II observability is also known as *multiple-experiment observability*. As a result, Type-I observability became known as *strong multiple-experiment observability* because it imposes a stronger condition than Type-II observability.

In situations where the *BCN*'s input can be controlled but the initial state *cannot* be reset, the initial state can only be determined through a single experiment, as it is not feasible to reset the initial state for repeated trials. This scenario is known as *single-experiment observability* [27]. Type-III observability formalizes this concept, stating that a *BCN* is observable if there exists an input sequence that can distinguish all possible initial states [5]. When the *BCN*'s input cannot be controlled, the observation problem is addressed by Type-IV observability. This type states that a *BCN* is observable if any sufficiently long input sequence can distinguish all distinct initial states [7]. This property is referred to as *arbitrary-experiment observability* because the initial state can only be determined by observing outputs without controlling inputs.

The initial state determination algorithm for Type-III observability involves running a *BCN* with an input sequence that distinguishes all initial states and then determining the initial state based on the output sequence [27]. However, we found that this algorithm fails for certain *BCNs*. In contrast, we developed a novel algorithm that can accurately determine their initial states in a single experiment. This suggests that single-experiment observability should not be defined by Type-III observability. Therefore, we propose a new type called Type-V observability to redefine single-experiment observability. With Type-V observability, it is possible to determine the initial states of a broader class of *BCNs* in just one experiment. Moreover, single-experiment observability is crucial for solving the identification problem of *BCNs* as the sufficient and necessary condition for identifiability involves a combination of controllability and it [5,20].

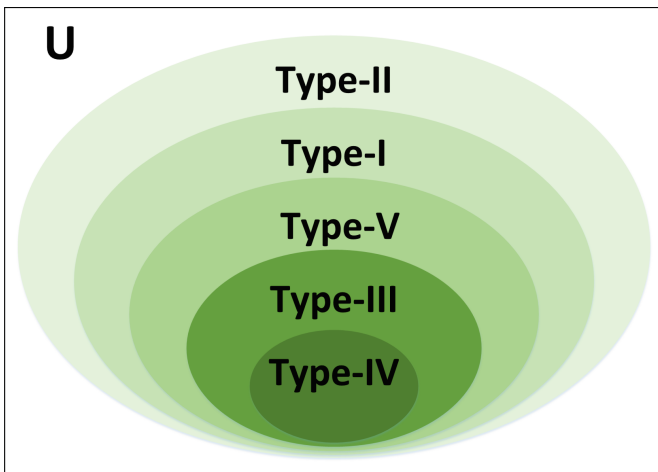


Fig. 1. Relations among different types of observability (elliptical areas labeled as Type-I, II, III, IV, and V) represent the sets of *BCNs* that are Type-(I, II, III, IV, and V) observable, respectively; area U represents the set of all *BCNs*).

The verification algorithm plays a crucial role in researching the observability of *BCNs*. For instance, to determine the initial state using the **Type-III observability** algorithm, we first need a verification algorithm to assess if the *BCN* satisfies **Type-III observability**. This verification algorithm also helps find an input sequence that can distinguish all initial states of the *BCN*. To advance the study of single-experiment observability, we further develop a verification algorithm for **Type-V observability** in this paper. This algorithm has lower computational complexity compared to those for **Type-III observability**. Comparing the verification algorithms for these two types of observability further highlights the advantages of **Type-V observability**.

Outline of the Paper. After the introduction, Sect. 2 presents the required notations and the formal definition of *BCNs*. In Sect. 3, **Type-V observability** is formally defined, and we present our novel solution algorithm for addressing the single-experiment observation problem in *BCNs*. Section 4 offers a verification algorithm for **Type-V observability** and analyzes its computational complexity. Finally, in Sect. 5, we summarize the findings and discuss potential avenues for future research.

2 Preliminaries

First, we introduce several essential notations in this section. It is important to note that, to maintain coherence in the notation and definitions used in our research, the notation and definitions of certain properties in this paper remain consistent with those used in our article [21] on the reconstructibility of *BCNs*.

- 2^A : the power set of set A ,
- $|A|$: the cardinality of set A ,
- \mathbb{B} : the set of Boolean values $\{0, 1\}$,
- \mathbb{B}^n : the n -dimensional Boolean vector space,
- \mathbb{T} : the set representing the discrete time domain which is denoted by the set of natural numbers,
- $v_{2^x}^i$: the x -dimensional Boolean vector whose decimal value is equal to i ,
- V_{2^x} : the set $\{v_{2^x}^0, \dots, v_{2^x}^{2^x-1}\}$ of Boolean vectors.

Next, we present the formal definition of Boolean control networks (*BCNs*). A *BCN* can be described by the following two equations [9]:

$$\begin{aligned} \mathbf{s}(t+1) &= \sigma(\mathbf{i}(t), \mathbf{s}(t)) \\ \mathbf{o}(t) &= \rho(\mathbf{s}(t)) \end{aligned} \tag{1}$$

where $t \in \mathbb{T}$; $\mathbf{i}(t) \in \mathbb{B}^\ell$, $\mathbf{s}(t) \in \mathbb{B}^m$, and $\mathbf{o}(t) \in \mathbb{B}^n$ denote the input vector, state vector, and output vector at time t , respectively; $\sigma : \mathbb{B}^\ell \times \mathbb{B}^m \mapsto \mathbb{B}^m$ and $\rho : \mathbb{B}^m \mapsto \mathbb{B}^n$ are logical functions determining the state and output of the network. Therefore, the relation between the inputs, states, and outputs of a *BCN* can be illustrated in Fig. 2, where $0, 1, \dots$ stand for time steps, $\mathbf{i}(0), \mathbf{i}(1), \dots$

represent inputs, $\mathbf{s}(0), \mathbf{s}(1), \dots$ represent states, $\mathbf{o}(0), \mathbf{o}(1), \dots$ represent outputs, and arrows represent dependence among inputs, states and outputs. Moreover, as we represent a x -dimensional Boolean vector whose decimal value is equal to i in the form $v_{2^x}^i$, the input set \mathbb{B}^ℓ , state set \mathbb{B}^m , and output set \mathbb{B}^n can be replaced by V_L, V_M , and V_N , respectively, where $L = 2^\ell, M = 2^m$, and $N = 2^n$.

To analyze the control-theoretic properties of *BCNs*, we define the following two classes of functions to represent the relationship between the input sequence, output sequence, and state sequence of a *BCN*.

$$\begin{aligned}
 F^{[t_0, t]} &: V_M \times (V_L)^{t-t_0+1} \mapsto (V_M)^{t-t_0+2}, \\
 F^{[t_0, t]}(\mathbf{s}(t_0), I[t_0, t]) &= \mathbf{s}(t_0) \dots \mathbf{s}(t+1), \\
 H^{[t_0, t]} &: V_M \times (V_L)^{t-t_0+1} \mapsto (V_N)^{t-t_0+2}, \\
 H^{[t_0, t]}(\mathbf{s}(t_0), I[t_0, t]) &= \mathbf{o}(t_0) \dots \mathbf{o}(t+1),
 \end{aligned} \tag{2}$$

where $t \geq t_0, I[t_0, t] \in (V_L)^{t-t_0+1}$ denotes the input sequence $i(t_0) \dots i(t)$ within the time interval $[t_0, t]$. For every state $\mathbf{s}(p)$ ($t_0 < p \leq t+1$) in the state sequence $\mathbf{s}(t_0) \dots \mathbf{s}(t+1)$, $\mathbf{s}(p) = \sigma(i(p-1), \mathbf{s}(p-1))$. For the output sequence $\mathbf{o}(t_0) \dots \mathbf{o}(t+1)$, every $\mathbf{o}(p)$ in it satisfies $\mathbf{o}(p) = \rho(\mathbf{s}(p))$.

Intuitively, the above two classes of functions describe the relationships among the input sequence $I[t_0, t]$, the state sequence $\mathbf{s}(t_0) \dots \mathbf{s}(t+1)$, and the output sequence $\mathbf{o}(t_0) \dots \mathbf{o}(t+1)$ within the time interval $[t_0, t+1]$. They generalize the two classes of functions given in [23] for observability, where only the special case of $F^{[0, t]}$ and $H^{[0, t]}$ is considered. These extensions will be helpful when we define Type-V observability. Moreover, $I[t_0, t]$ would be replaced by $I[t]$ when $t_0 = 0$ for the sake of conciseness in the following sections.

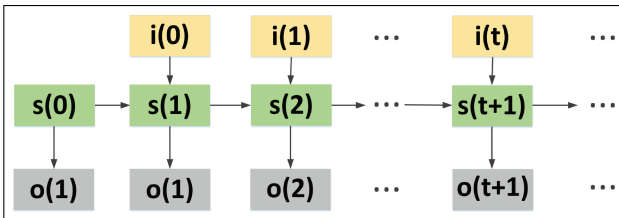


Fig. 2. An illustration of the relationships between inputs, states, and outputs of *BCNs*.

3 Observability of *BCNs*

In this section, we redefine the concept of single-experiment observability for *BCNs* by introducing and formally defining Type-V observability.

3.1 Problems in Existing Definitions

Before doing so, we will first present the existing definition of single-experiment observability (Type-III observability) and explain why it is considered incorrect.

Definition 1 (Type-III observability [5]). *A BCN satisfies Type-III observability if there exists an input sequence $I[p]$ for some $p \in \mathbb{T}$, such that for any two distinct states $s, s' \in V_M$, $H^{[0,p]}(s', I[p]) \neq H^{[0,p]}(s, I[p])$.*

Certain researchers consider Type-III observability as the single-experiment observability for BCNs because they argue that the following straightforward algorithm is the only method to determine a BCN's initial state in a single experiment (this conclusion is inferred from [25]). In this paper, we present all solution algorithms for the observation problem of BCNs by outlining their primary steps, as demonstrated in the following example.

(Step 1) Feed a BCN an input sequence $I[p]$ that distinguishes all 2^m distinct initial states of this BCN, and execute the network to generate its output sequence $o(0) \dots o(p+1)$.

(Step 2) Return the BCN's initial state $s(0)$ that satisfies $H^{[0,p]}(s(0), I[p]) = o(0) \dots o(p+1)$.

The algorithm described above determines the initial state $s(0)$ of the BCN using the input sequence $I[p]$ and the output sequence $o(0) \dots o(p+1)$. This algorithm relies on the input sequence $I[p]$ distinguishing all 2^m distinct initial states of the BCN. For this algorithm to be effective, the condition $H^{[0,p]}(s', I[p]) \neq H^{[0,p]}(s, I[p])$ must hold for any two distinct states s and s' of the BCN. However, we found certain BCNs (e.g., Table 1) where the existing algorithm fails to accurately determine initial states. As a result, we have developed a new algorithm that successfully determines the initial states of these BCNs in a single experiment.

Example 1. The BCN as shown in Table 1 does not satisfy Type-III observability, because for any $p \in \mathbb{T}$,

- for any input sequence $I[p]$ with the prefix v_2^0 , e.g., $v_2^0 v_2^1$,
 $H^{[0,p]}(v_8^1, I[p]) = H^{[0,p]}(v_8^2, I[p])$;
- for any input sequence $I[p]$ with the prefix v_2^1 , e.g., $v_2^1 v_2^1$,
 $H^{[0,p]}(v_8^6, I[p]) = H^{[0,p]}(v_8^7, I[p])$.

Therefore, for any $p \in \mathbb{T}$, there is not any input sequence $I[p]$ which satisfies that for any two distinct states $s, s' \in V_8$, $H^{[0,p]}(s', I[p]) \neq H^{[0,p]}(s, I[p])$.

Table 1. A *BCN* used to illustrate various concepts, showing how the value of $s(t+1)$ is determined by the values of $s(t)$ (row dimension) and $i(t)$ (column dimension), and how the value of $o(t)$ is determined solely by the value of $s(t)$, being independent of the value of $i(t)$.

$s(t)$	v_8^0	v_8^1	v_8^2	v_8^3	v_8^4	v_8^5	v_8^6	v_8^7	$i(t)$
$s(t+1)$	v_8^2	v_8^3	v_8^3	v_8^6	v_8^1	v_8^3	v_8^4	v_8^3	v_2^0
	v_8^7	v_8^5	v_8^1	v_8^4	v_8^2	v_8^6	v_8^0	v_8^0	v_2^1
$o(t)$	v_4^0	v_4^1	v_4^1	v_4^1	v_4^2	v_4^2	v_4^3	v_4^3	\cdot

3.2 Proposal and Justification of a New Definition

To define single-experiment observability for *BCNs*, we introduce the concept of the state set $S_{ps}(t)$, which comprises all possible valuations of the state $s(t)$ that can be inferred for a *BCN* at time step t . In other words, the state set $S_{ps}(t)$ is derived from the historical information of the *BCN*'s inputs and outputs up to time t . We illustrate the process of calculating $S_{ps}(t)$ for a *BCN* through the following steps.

First, we define the function $\zeta(S, i, o)$ to represent how the state set $S_{ps}(t)$ for a *BCN* is calculated using its state set $S_{ps}(t-1)$, input $i(t-1)$, and output $o(t)$. However, before defining the function $\zeta(S, i, o)$, we need to introduce the function $\xi(i, s)$:

$$\begin{aligned} \xi &: (V_L \cup \{\varepsilon\}) \times V_M \mapsto V_M, \\ \xi(i, s) &= \begin{cases} \sigma(i, s) & i \neq \varepsilon \\ s & i = \varepsilon \end{cases}. \end{aligned} \tag{3}$$

The function $\xi(i, s)$ is defined to capture how the state s of the *BCN* is affected by its input i . Unlike the *BCN*'s updating function $s(t+1) = \sigma(i(t), s(t))$, this function $\xi(i, s)$ specifically captures how the state s changes when the input i is absent (i.e., $i = \varepsilon$).

Next, we define the function $\zeta(S, i, o)$ as follows:

$$\begin{aligned} \zeta &: 2^{V_M} \times (V_L \cup \{\varepsilon\}) \times (V_N \cup \{\varepsilon\}) \mapsto 2^{V_M} \\ \zeta(S, i, o) &= \begin{cases} \{\xi(i, s) \mid s \in S, \rho(\xi(i, s)) = o\} & o \neq \varepsilon \\ \{\xi(i, s) \mid s \in S\} & o = \varepsilon \end{cases} \end{aligned} \tag{4}$$

where ε presents the absence of input or output.

The function $\zeta(S, i, o)$ outlines how to calculate the state set $S_{ps}(t)$ for a *BCN*. When $t = 0$, we can only utilize the initial output $o(0)$ to determine the state set $S_{ps}(t)$ for the *BCN*. Therefore, the *BCN*'s state set $S_{ps}(t) = \zeta(V_M, \varepsilon, o(0))$, which means $S_{ps}(t)$ (or $S_{ps}(0)$) includes all initial states that can produce the initial output $o(0)$. When $t > 0$, the *BCN*'s state set $S_{ps}(t)$ is determined by its state set $S_{ps}(t-1)$, input $i(t-1)$, and output $o(t)$. Hence, the set $S_{ps}(t) = \zeta(S_{ps}(t-1), i(t-1), o(t))$, which implies that for every state s in $S_{ps}(t)$, there exists a state s' in $S_{ps}(t-1)$ such that $s = \sigma(i(t-1), s')$. Furthermore, all states belonging to the state set $S_{ps}(t)$ should be capable of generating the output $o(t)$.

Second, building upon the previous concept, we recursively define the following class of functions $G^{[t]}(\mathbb{I}[t-1], \mathbf{o}(0) \dots \mathbf{o}(t))$ to represent how to determine the set $\mathcal{S}_{ps}(t)$ for a *BCN* by analyzing its input sequence $\mathbb{I}[t-1]$ and output sequence $\mathbf{o}(0) \dots \mathbf{o}(t)$. In other words, we define $\mathcal{S}_{ps}(t) = G^{[t]}(\mathbb{I}[t-1], \mathbf{o}(0) \dots \mathbf{o}(t))$. This recursive definition allows us to determine the state set $\mathcal{S}_{ps}(t)$ at each time step t by considering the input sequence $\mathbb{I}[t-1]$ and the output sequence $\mathbf{o}(0) \dots \mathbf{o}(t)$ of the *BCN*.

$$G^{[t]} : V_L^t \times V_N^{t+1} \mapsto 2^{V_M} \quad (5)$$

These functions satisfy the following conditions.

- When $t = 0$, $\mathbb{I}[t-1] = \varepsilon$,

$$G^{[t]}(\mathbb{I}[t-1], \mathbf{o}(0) \dots \mathbf{o}(t)) = \zeta(V_M, \varepsilon, \mathbf{o}(0)).$$

- When $t > 0$,

$$G^{[t]}(\mathbb{I}[t-1], \mathbf{o}(0) \dots \mathbf{o}(t)) = \zeta(G^{[t-1]}(\mathbb{I}[t-2], \mathbf{o}(0) \dots \mathbf{o}(t-1)), \mathbb{I}[t-1], \mathbf{o}(t)).$$

Third, with the introduction of the state set $\mathcal{S}_{ps}(t)$, we can address the single-experiment observation problem of *BCNs* more clearly. In this problem, we perform a single experiment to generate a set of input-output sequences from a *BCN*, aiming to determine the *BCN*'s initial state. We conclude that the state $\mathbf{s}(t)$ of a *BCN* can be determined within a finite number of time steps k by conducting a single experiment if and only if the following conditions are satisfied.

- $|\mathcal{S}_{ps}(t+k)| = 1$, i.e. the *BCN*'s $\mathbf{s}(t+k)$ is determined.
- For every t_0 such that $t+1 \leq t_0 \leq t+k$, and for every state \mathbf{s} in $\mathcal{S}_{ps}(t_0)$, there exists only one state \mathbf{s}' in $\mathcal{S}_{ps}(t_0-1)$ that satisfies $\mathbf{s} = \sigma(\mathbb{I}[t_0-1], \mathbf{s}')$, ensuring that the *BCN*'s state $\mathbf{s}(t_0-1)$ can be uniquely determined by its state $\mathbf{s}(t_0)$ and input $\mathbb{I}[t_0-1]$. By following this idea, the *BCN*'s state $\mathbf{s}(t)$ can be uniquely determined step by step, starting from its state $\mathbf{s}(t+k)$ and utilizing the input sequence $\mathbb{I}[t, t+k-1]$.

These conditions are both necessary for determining the initial state of a *BCN* in a single experiment.

- Necessity of the first condition: If the state $\mathbf{s}(t)$ of a *BCN* is determined, it implies that its state $\mathbf{s}(t+k)$ can be determined step by step using the update function σ and the input sequence $\mathbb{I}[t, t+k-1]$. Therefore, if the *BCN*'s state $\mathbf{s}(t+k)$ cannot be guaranteed to be determined, it is impossible to ensure that the state $\mathbf{s}(t)$ can be determined in k time steps. The determination of the initial state relies on the availability of the state at time $t+k$, and without it, the determination of the state at time t becomes uncertain.
- Necessity of the second condition: If the second condition is not guaranteed to be met, it implies that there is no assurance that the *BCN*'s state $\mathbf{s}(t)$ can be uniquely determined step by step using the state $\mathbf{s}(t+k)$ and the input sequence $\mathbb{I}[t, t+k-1]$. In this case, there may be multiple possible

previous states leading to the same current state, introducing ambiguity in the backward propagation. Consequently, even if the state $\mathbf{s}(t + k)$ can be determined, there is no guarantee that the *BCN*'s state $\mathbf{s}(t)$ can be uniquely determined in k time steps through a single experiment.

Based on the aforementioned conclusion, we introduce the concept of the set of state sets, denoted as $Set_{\mathcal{S}}(k)$. This set represents the collection of state sets where, for each state set S belonging to $Set_{\mathcal{S}}(k)$, it satisfies the condition that it takes k time steps to determine the state $\mathbf{s}(t)$ of a *BCN* through a single experiment when its state set $S_{ps}(t)$ is equal to S . The determination of $Set_{\mathcal{S}}(k)$ can be achieved through the following recursive steps.

- When $k = 0$, then $Set_{\mathcal{S}}(k) = \{S \in 2^{V_M} \mid |S| = 1\}$.
- When $k > 0$, then $Set_{\mathcal{S}}(k) = \{S \in (2^{V_M} - \bigcup_{p=0}^{k-1} Set_{\mathcal{S}}(p)) \mid \exists i \in V_L \cdot (|\zeta(S, i, \varepsilon)| = |S|) \& (\forall o \in V_N \cdot \exists p \leq (k - 1) \cdot \zeta(S, i, o) \in Set_{\mathcal{S}}(p))\}$.

Intuitively, when $|S| = |S_{ps}(t)| = 1$, i.e. $\mathbf{s}(t)$ is determined, we need 0 time step to determine $\mathbf{s}(t)$. Thus, we set $Set_{\mathcal{S}}(k) = \{S \in 2^{V_M} \mid |S| = 1\}$ when $k = 0$. When $k > 0$, we use the sets $Set_{\mathcal{S}}(0), \dots, Set_{\mathcal{S}}(k - 1)$ that have been defined to define the set $Set_{\mathcal{S}}(k)$. Firstly, as the set $Set_{\mathcal{S}}(k)$ should not intersect with the sets $Set_{\mathcal{S}}(0), \dots, Set_{\mathcal{S}}(k - 1)$, we have for every state set $S \in Set_{\mathcal{S}}(k)$, the condition $S \in (2^{V_M} - \bigcup_{p=0}^{k-1} Set_{\mathcal{S}}(p))$ should be met. Secondly, as the *BCN*'s state $\mathbf{s}(t)$ should be determined by its state $\mathbf{s}(t + 1)$ and input $i(t)$, those following conditions also need to be satisfied as we discussed in the previous paragraph.

We can define the function $\Gamma(S)$ to represent the number of time steps required to determine the state $\mathbf{s}(t)$ of a *BCN* through a single experiment, given that its state set $S_{ps}(t) = S$. The function $\Gamma(S)$ provides the time step count needed for state determination in this scenario.

$$\Gamma : (2^{V_M} - \{\emptyset\}) \mapsto (\mathbb{T} \cup \{\infty\}) \tag{6}$$

satisfies the following conditions:

- If there exists a finite number k which satisfies that $S \in Set_{\mathcal{S}}(k)$, $\Gamma(S) = k$.
- Otherwise, $\Gamma(S) = \infty$.

In the previous paragraphs, we explored the determination of whether the state $\mathbf{s}(t)$ of a *BCN* can be established within a finite number of time steps through a single experiment, given its state set $S_{ps}(t)$. Extending this concept, we can consider all possible $S_{ps}(0)$ for a *BCN* to determine if its initial state $\mathbf{s}(0)$ can be guaranteed to be determined in one experiment. This leads us to define the **Type-V observability** as the single-experiment observability of *BCNs*.

Definition 2 (Type-V Observability). A *BCN* satisfies **Type-V observability** if for every possible $S_{ps}(0)$ of this *BCN*, $\Gamma(S_{ps}(0)) \neq \infty$.

In order to address the single-experiment observation problem, we present a novel solution algorithm. For this purpose, we define the function $\psi(S)$ to

represent the set of inputs available at time t when executing the algorithm to determine the initial state of a *BCN*, given that the *BCN*'s $S_{ps}(t)$ is equal to S . The function $\psi(S)$ provides the set of selectable inputs during the execution of the algorithm for initial state determination.

$$\begin{aligned} \psi &: (2^{V_M} - \emptyset) \mapsto 2^{V_L} \\ \psi(S) &= \{i \in V_L \mid |\zeta(S, i, \varepsilon)| = |S|, \forall o \in V_N \\ &\quad \zeta(S, i, o) \neq \emptyset \rightarrow \Gamma(\zeta(S, i, o)) \neq \infty\} \end{aligned} \quad (7)$$

The definition of function $\psi(S)$ is obtained from the conditions mentioned in the definition of $\Gamma(S)$. Then, for a *BCN* with **Type-V observability**, we provide the following algorithm with four main steps to determine its initial state.

(Step 1) Obtain the state set $S_{ps}(0)$ of this *BCN* by its initial output $o(0)$, i.e.

$S_{ps}(0) := \zeta(V_M, \varepsilon, o(0))$, and set the set variable S by $S_{ps}(0)$, i.e. $S := S_{ps}(0)$.

(Step 2) Select an input i which satisfies the formula

$$\max_{o' \in \{o \mid \zeta(S, i, o) \neq \emptyset\}} \Gamma(\zeta(S, i, o')) + 1 = \Gamma(S)$$

from $\psi(S)$, and simulate the *BCN* with the input i to generate a new output $o(t)$.

(Step 3) Determine the new $S_{ps}(t)$ by the input i , output $o(t)$, and set variable S , i.e. $S_{ps}(t) := \zeta(S, i, o(t))$, and update the set variable S by $S_{ps}(t)$, i.e. $S := S_{ps}(t)$.

(Step 4) If $|S| = 1$, then return $s(0)$ satisfying $H^{[0, t-1]}(s(0), I[t-1]) = o(0) \dots o(t)$ as the initial state of this *BCN*. Otherwise, update t , i.e. $t := t+1$, and go to Step 2.

As discussed in the previous paragraphs, the condition $|\zeta(S, i, \varepsilon)| = |S|$ is consistently satisfied at each time step. This guarantees that the initial state $s(0)$ of the studied *BCN* can be determined based on the input sequence $I[t-1]$ and the output sequence $o(0) \dots o(t)$, once the state $s(t)$ is determined. Furthermore, the equation

$$\max_{o' \in \{o \mid \zeta(S, i, o) \neq \emptyset\}} \Gamma(\zeta(S, i, o')) + 1 = \Gamma(S)$$

being satisfied at each time step ensures that the state $s(t)$ of the *BCN* can be determined. These observations highlight the significance of these conditions in enabling the determination of the state of the *BCN* at each time step, ultimately leading to the determination of the initial state for a *BCN* with **Type-V observability**.

Compared to the algorithm for solving the single-experiment observation problem associated with **Type-III observability**, the algorithm described above differs in that it does not require an input sequence capable of distinguishing all initial states. Instead, it relies on a strategy to determine the input $i(t)$ of the *BCN* at each time step based on the known inputs and outputs of the *BCN*. In other words, in the above algorithm, the *BCN*'s inputs $i(t)$ are not predetermined

but determined dynamically based on the *BCN*'s state set $S_{ps}(t)$, which, in turn, is determined by the *BCN*'s inputs and outputs. This characteristic allows the above algorithm to determine the initial states of a broader range of *BCNs* (including cases illustrated in Table 1) compared to the algorithm associated with Type-III observability. This distinction emphasizes why the above algorithm can effectively determine the initial states of various *BCNs* by leveraging the dynamic determination of inputs based on the *BCN*'s state set, contributing to a more extensive applicability in practice.

Example 2. Continuing with the *BCN* shown in Table 1, we have

- $\Gamma(S_{ps}(0)) = 0$, when $S_{ps}(0) = \zeta(V_M, \varepsilon, \mathbf{o}(0)) = \zeta(V_8, \varepsilon, v_4^0) = \{v_8^0\}$;
- $\Gamma(S_{ps}(0)) = 2$, when $S_{ps}(0) = \zeta(V_M, \varepsilon, \mathbf{o}(0)) = \zeta(V_8, \varepsilon, v_4^1) = \{v_8^1, v_8^2, v_8^3\}$;
- $\Gamma(S_{ps}(0)) = 1$, when $S_{ps}(0) = \zeta(V_M, \varepsilon, \mathbf{o}(0)) = \zeta(V_8, \varepsilon, v_4^2) = \{v_8^4, v_8^5\}$;
- $\Gamma(S_{ps}(0)) = 1$, when $S_{ps}(0) = \zeta(V_M, \varepsilon, \mathbf{o}(0)) = \zeta(V_8, \varepsilon, v_4^3) = \{v_8^6, v_8^7\}$.

Therefore, this *BCN* satisfies Type-V observability and the initial state of it can be determined by the above algorithm.

To further demonstrate that Type-V observability is the appropriate definition for single-experiment observability in *BCNs*, we can provide a formal comparison with Type-III observability through the proof of the following theorem.

Theorem 1. Type-III observability *implies* Type-V observability.

Proof. We present a proposition that for a set $S_{ps}(x)$ of a *BCN*'s possible state valuations at time x , if there exists an input sequence $l[x, p]$ for some $p \geq x$, such that for any two distinct states $\mathbf{s}(x), \mathbf{s}'(x) \in S_{ps}(x)$, $H^{[x,p]}(\mathbf{s}'(x), l[x, p]) \neq H^{[x,p]}(\mathbf{s}(x), l[x, p])$ holds, then $\Gamma(S_{ps}(x)) \neq \infty$. We prove this by induction.

- When $p = x$, if for any two distinct states $\mathbf{s}(x), \mathbf{s}'(x) \in S_{ps}(x)$, we have $H^{[x,p]}(\mathbf{s}'(x), l[x, p]) \neq H^{[x,p]}(\mathbf{s}(x), l[x, p])$, then the input $i(x)$ which is equal to $l[x, p]$ satisfies
 - $|\zeta(S_{ps}(x), i(x), \varepsilon)| = |S_{ps}(x)|$, and
 - for every non-empty $\zeta(S_{ps}(x), i(x), \mathbf{o}(x+1))$, $\Gamma(\zeta(S_{ps}(x), i(x), \mathbf{o}(x+1))) = 0$ holds.

Therefore, $\Gamma(S_{ps}(x)) = 1$, i.e. the above proposition holds when $p = x$.

- Assuming that the above proposition holds when $p = x, \dots, (x+k)$. Then, when $p = x+k+1$, if for any distinct states $\mathbf{s}(x), \mathbf{s}'(x) \in S_{ps}(x)$, we have $H^{[x,p]}(\mathbf{s}'(x), l[x, p]) \neq H^{[x,p]}(\mathbf{s}(x), l[x, p])$, then for $S_{ps}(x)$, the input $i(x)$ which is the first input of $l[x, p]$ satisfies
 - $|\zeta(S_{ps}(x), i(x), \varepsilon)| = |S_{ps}(x)|$, and
 - for every non-empty $\zeta(S_{ps}(x), i(x), \mathbf{o}(x+1))$, $\Gamma(\zeta(S_{ps}(x), i(x), \mathbf{o}(x+1))) \neq \infty$,

i.e. $\Gamma(S_{ps}(x)) \neq \infty$. Thus the proposition holds when $p = x+k+1$ if it holds when $p = x, \dots, (x+k)$.

Therefore, the above proposition holds for any $p \geq x$. Secondly, if a *BCN* satisfies **Type-III observability**, then there exists an input sequence $I[p]$ for some $p > 0$, such that for any two distinct states $s(0), s'(0) \in V_M$, $H^{[0,p]}(s'(0), I[p]) \neq H^{[0,p]}(s(0), I[p])$ holds, thus for every possible $S_{ps}(0)$, $\Gamma(S_{ps}(0)) \neq \infty$. Therefore, **Type-III observability** implies **Type-V observability**.

The aforementioned theorem demonstrates that any *BCN* satisfying **Type-III observability** also satisfies **Type-V observability**. However, the *BCN* presented in Table 1 illustrates that not all *BCNs* satisfying **Type-V observability** would necessarily satisfy **Type-III observability**. Therefore, we can conclude that for *BCNs*, **Type-V observability** is comparatively easier to satisfy than **Type-III observability**. Moreover, by considering the discussion on the interrelationships among the four properties of observability (**Type-I, II, III & IV**) presented in paper [27], we can ascertain the overall relationship among all five types of observability, as depicted in Fig. 1 in Sect. 1).

Remark. It is worth noting that another type of observability known as **Output-Feedback observability** ([8]) has been proposed to capture the property required for a *BCN* to enable the execution of an output-feedback algorithm to determine the initial state. It is equivalent to **Type-V observability**. However, it is not mentioned in [8] that **Output-Feedback observability** represents a new definition of single-experiment observability for *BCNs*. In contrast, our work provides a detailed discussion on why **Type-V observability** precisely captures the concept of single-experiment observability for *BCNs*. This constitutes the first part of our contribution, where we thoroughly explain how **Type-V observability** addresses the requirement of determining the initial state through a single experiment. Furthermore, observability plays a pivotal role in addressing the problem of identifying *BCNs* because the integration of single-experiment observability and controllability is a sufficient condition for identifiability [5]. Therefore, clarifying the concept of single-experiment observability in this paper is also crucial for solving the identification problem of *BCNs*.

4 Verification of Type-V Observability

In this section, we present a verification algorithm to establish the **Type-V observability** of the considered *BCN*. The algorithm aims to determine whether every possible $S_{ps}(0)$ of the *BCN* satisfies $\Gamma(S_{ps}(0)) \neq \infty$. Additionally, it provides a strategy for determining the input to execute the *BCN* at each time step, enabling the determination of the *BCN*'s initial state using the algorithm introduced in Sect. 3. To solve the verification problem, we first define an input-labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ for *BCNs*.

Definition 3 (Input-labelled graph). Let \mathcal{V} , \mathcal{E} and \mathcal{L} be the vertex set, edge set, and labelling function of an input-labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$. \mathcal{G} is called the input-labelled graph of a *BCN* if

$$\begin{aligned}
- \mathcal{V} &= \{S \in (\bigcup_{\mathbf{o} \in V_N} 2^{\zeta(V_M, \varepsilon, \mathbf{o})} - \emptyset) \mid \Gamma(S) \neq \infty\}; \\
- \mathcal{E} &= \{(S_1, S_2) \in \mathcal{V} \times \mathcal{V} \mid |S_1| > 1, S_2 \in \{\zeta(S_1, i, \mathbf{o}) \mid i \in \psi(S_1), \mathbf{o} \in V_N\}\}; \\
- \mathcal{L} : \mathcal{E} &\mapsto 2^{V_L}, \mathcal{L}(S_1, S_2) = \{i \in \psi(S_1) \mid S_2 \in \{\zeta(S_1, i, \mathbf{o}) \mid \mathbf{o} \in V_N\}\}.
\end{aligned}$$

Intuitively, in the input-labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ of a *BCN*, the vertex set \mathcal{V} represents the set of all state sets that satisfy $\Gamma(S) \neq \infty$. Furthermore, for any two distinct states $s, s' \in S$, they yield the same output. The edge set \mathcal{E} captures the relationships between the state sets belonging to \mathcal{V} . The labelling function \mathcal{L} assigns a set of inputs to each edge $e \in \mathcal{E}$ in the graph. In this way, the **Type-V observability** of a *BCN* can be verified by constructing its input-labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ and examining whether every possible $S_{ps}(0)$ belongs to the vertex set \mathcal{V} . Additionally, the strategy for determining the input to execute the *BCN* at each time step can be derived from its input-labelled graph.

After presenting the main idea of our verification algorithm, we now provide additional details on constructing and checking the input-labelled graph for a *BCN*. Firstly, considering that the function $\Gamma(S)$ is recursively defined, we construct the vertices consisting of smaller state sets before constructing those consisting of larger state sets in the process of constructing the input-labelled graph. Secondly, we introduce two lemmas regarding the functions $\Gamma(S)$ and $\psi(S)$ to provide further insight into the design of our algorithm.

Lemma 1. *For any two non-empty state sets S^1 and S^2 , if $S^1 \subseteq S^2$ and $\Gamma(S^2) \neq \infty$, then $\Gamma(S^1) \neq \infty$.*

Lemma 2. *For any two non-empty state sets S^1 and S^2 , if $S^1 \subseteq S^2$ and $\Gamma(S^2) \neq \infty$, then $\psi(S^2) \subseteq \psi(S^1)$.*

Due to space constraints, we omit the proofs of the aforementioned lemmas. Based on Lemma 1, if we encounter a set of states S that does not belong to \mathcal{V} , it implies the existence of an output $\mathbf{o} \in V_N$ such that $S \subseteq \zeta(V_M, \varepsilon, \mathbf{o})$, leading to $\Gamma(\zeta(V_M, \varepsilon, \mathbf{o})) = \infty$. Consequently, the *BCN* fails to satisfy **Type-V observability**. In such cases, there is no need to continue constructing the input-labelled graph, as the verification for the *BCN*'s **Type-V observability** has already been established.

Moreover, Lemma 2 demonstrates that once we have determined $\psi(S')$ for every $S' \subset S$, we can approximate the scope of $\psi(S)$ for the set S by calculating $\bigcap_{S' \subset S} \psi(S')$. This allows for an easier determination of $\Gamma(S)$.

After introducing the principles of our designed verification algorithm, we now present its detailed description in Algorithm 1. To illustrate how Algorithm 1 works, we use the *BCN* provided in Table 1 as an example to demonstrate the steps of the algorithm. In summary, Algorithm 1 constructs the input-labelled graph (depicted in Fig. 3) of the *BCN* in a layer-by-layer manner, starting from the bottom and moving upwards. We now provide additional details about Algorithm 1.

Let us first explain Algorithm 2, which is called within Algorithm 1. Its purpose is to construct the set of vertices based on the number z of states that

the vertices contain. Since any two distinct states in a vertex should produce the same output, we use the formula

$$Ver_arr = \{S \in \bigcup_{o \in V_N} 2^{\zeta(V_M, \varepsilon, o)} \mid |S| = z\}$$

to obtain the set Ver_arr (line 2). If Ver_arr is empty, Algorithm 2 returns Null (lines 3–4). If Ver_arr is not empty, Algorithm 2 returns Ver_arr (lines 5–6). For the BCN in Table 1, when $z = 1$, the set of vertices constructed by Algorithm 2 contains the vertices at the bottom of the graph shown in Fig. 3. When $z = 4$, Algorithm 2 returns Null.

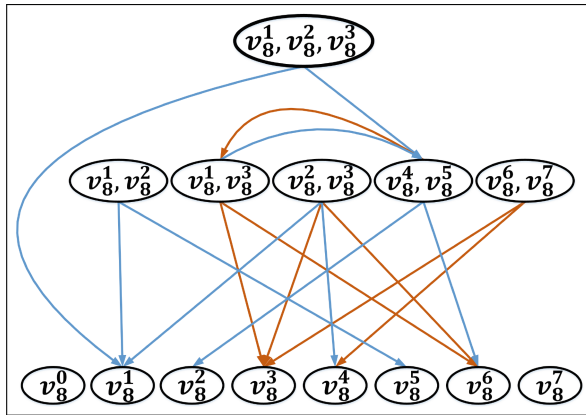


Fig. 3. The input-labelled graph, where the orange and blue edges are labelled with $\{v_2^0\}$ and $\{v_2^1\}$, respectively. (Color figure online)

Next, we introduce Algorithm 1. This algorithm calls Algorithm 2 to construct the set Ver_arr of vertices based on the number z of states that the vertices contain. It then checks whether the vertices in Ver_arr belong to the input-labelled graph of the BCN or not. When z is equal to 1, it is known directly that the vertices returned by Algorithm 2 belong to the input-labelled graph of the BCN based on the definition of $\Gamma(S)$. Therefore, Algorithm 1 starts checking Ver_arr when z is equal to 2 (lines 3–4). If Algorithm 2 returns Null, it indicates that all possible initial state sets $S(0)$ of the BCN have been checked, implying that the BCN satisfies Type-V observability. In this case, Algorithm 1 returns the input-labelled graph of the BCN (lines 5, 22–24). In_arr represents the input set to be checked for a vertex $Ver_arr[i]$. When z is greater than 2, Algorithm 1 uses the formula

$$In_arr = \bigcap_{S \in Ver_arr[i]} \psi(S)$$

Algorithm 1. Verification algorithm for Type-V observability

Input: The updating rules of a *BCN*
Output: The input-labelled graph of this *BCN*

```

1: integer  $i, z = 1$ 
2: array  $Ver\_arr[ ], In\_arr[ ]$ 
3:  $Ver\_arr = \text{constructvertices}(z)$ 
4:  $Ver\_arr = \text{constructvertices}(++z)$ 
5: while ( $Ver\_arr \neq \text{Null}$ ) do
6:   for ( $i = 0; i < \text{arraysize}(Ver\_arr); i++$ ) do
7:     if ( $z == 2$ ) then
8:        $In\_arr = V_L$ 
9:     else
10:       $In\_arr = \bigcap_{S \subset Ver\_arr[i]} \psi(S)$ 
11:    end if
12:    if ( $In\_arr == \emptyset$ ) then
13:      Return Null
14:    end if
15:    Get  $\psi(Ver\_arr[i])$  by  $In\_arr$  and existing vertices
16:    if ( $\psi(Ver\_arr[i]) \neq \emptyset$ ) then
17:      Build edges for  $Ver\_arr[i]$ 
18:    else
19:      Return Null
20:    end if
21:  end for
22:   $Ver\_arr = \text{constructvertices}(++z)$ 
23: end while
24: Return  $\text{constructvertices}(--z)$ 

```

to obtain In_arr (lines 7–11) based on Lemma 2. Then, Algorithm 1 uses In_arr to check the vertex $Ver_arr[i]$. If In_arr is empty, it indicates that the *BCN* does not satisfy Type-V observability according to Lemma 1. In this case, Algorithm 1 returns **Null** (lines 12–14). If In_arr is not empty, Algorithm 1 checks the vertex $Ver_arr[i]$ and creates edges for it, as depicted in Fig. 3 (lines 15–20). From Fig. 3, we can observe that every possible initial state set $S_{ps}(0)$ of the *BCN* in Table 1 belongs to the vertex set \mathcal{V} of its input-labelled graph. In Sect. 3, we established that this *BCN* satisfies Type-V observability. Thus, this example confirms the correctness of our algorithm.

For a *BCN* with Type-V observability, with its input-labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$, a necessary strategy for determining its initial state can be easily obtained. We take the *BCN* in Table 1 as an example to illustrate this. For a *BCN* with Type-V observability, the input-labelled graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{L})$ provides a crucial strategy to determine its initial state. To illustrate this, we consider the *BCN* presented in Table 1.

Example 3. For this *BCN*, we first calculate the $\Gamma(S)$ value for every state set S in its input-labelled graph, starting from the bottom and moving upwards.

Algorithm 2. `constructvertices(integer z)`

Input: The number of states z
Output: The vertices with z states producing the same output

```

1: array  $Ver\_arr[]$ 
2:  $Ver\_arr = \{S \in \bigcup_{o \in V_N} 2^{\zeta(V_M, \varepsilon, o)} \mid |S| = z\}$ 
3: if ( $Ver\_arr == \emptyset$ ) then
4:   Return Null
5: else
6:   Return  $Ver\_arr$ 
7: end if
    
```

The resulting values are shown in Table 2. Next, we construct a strategy for this *BCN*, which is presented in Table 3. This strategy ensures that for every possible state set $S(t)$, the chosen input $i(t)$ satisfies the following equation:

$$\max_{o' \in \{o \mid \zeta(S_{ps}(t), i(t), o) \neq \emptyset\}} \Gamma(\zeta(S_{ps}(t), i(t), o')) + 1 = \Gamma(S(t))$$

This equation guarantees that the maximum value of $\Gamma(\zeta(S_{ps}(t), i(t), o'))$ for all possible output values o' that can be deduced from the state set $S_{ps}(t)$ using input $i(t)$ is equal to $\Gamma(S(t))$ minus one.

Table 2. The $\Gamma(S)$ value of the state set S ($|S| > 1$) in the Fig. 3.

S	$\{v_8^1, v_8^2\}$	$\{v_8^1, v_8^3\}$	$\{v_8^2, v_8^3\}$	$\{v_8^4, v_8^5\}$	$\{v_8^6, v_8^7\}$	$\{v_8^1, v_8^2, v_8^3\}$
$\Gamma(S)$	1	1	1	1	1	2

Table 3. A strategy for the *BCN* in Table 1.

$S_{ps}(t)$	$\{v_8^1, v_8^2\}$	$\{v_8^1, v_8^3\}$	$\{v_8^2, v_8^3\}$	$\{v_8^4, v_8^5\}$	$\{v_8^6, v_8^7\}$	$\{v_8^1, v_8^2, v_8^3\}$
$i(t)$	v_2^1	v_2^0	v_2^1	v_2^1	v_2^0	v_2^0

After presenting our algorithm for verifying the Type-V observability of *BCNs*, we proceed to analyze the computational complexity of this algorithm. Given a *BCN* with ℓ input-nodes, m state-nodes, and n output-nodes, the input-labelled graph of the *BCN* can have up to $2^{2^m - 1}$ vertices. This occurs when the *BCN* satisfies Type-V observability, and there exists an output node $o \in V_N$ such that $|\zeta(V_M, \varepsilon, o)| = 2^m - 1$. Furthermore, each vertex needs to be checked at most 2^ℓ times. Hence, the upper bound of the computational complexity for Algorithm 1 is $O(2^{2^m + \ell - 1})$. In contrast, the computational complexity of the verification

algorithms for Type-III observability is $2^{2^{2m-1}\ell}$ according to [25]. This comparison of computational complexity highlights the advantages of Type-V observability over Type-III observability when solving the single-experiment observation problem in *BCNs*. Moreover, due to our utilization of Lemma 1, the computational complexity of Algorithm 1 matches the lower bound for any algorithm aiming to verify Type-V observability, which is $O(2^\ell)$. This lower bound is achieved when the *BCN* under study does not satisfy Type-V observability, and the first state set $Ver_arr[i]$ to be checked does not satisfy $\psi(Ver_arr[i]) \neq \emptyset$. While for the Output-Feedback observability, which is equivalent to Type-V observability but defined differently, was investigated in [8], there appears to be a lack of research on its verification algorithm. Thus, our proposed algorithm further advances the study of Output-Feedback observability for *BCNs*.

5 Conclusions

In this paper, we have redefined single-experiment observability for *BCNs*, and proposed an advanced verification algorithm tailored to this new observability form. Our algorithm efficiently identifies initial states in *BCNs* using a single experiment, surpassing existing methods. Our study provides a solution to the challenging problem of single-experiment observation in *BCNs*. However, verifying single-experiment observability for large-scale *BCNs* remains challenging due to computational complexity. To address this, it is a viable approach to segment a large-scale network into components [17, 22] and subsequently investigate its observability, with existing research having achieved relevant results within the four existing categories of observability [26]. Consequently, we will pursue this line of research in an attempt to resolve the problem. Additionally, engineering the verification algorithm we designed and developing a toolkit to facilitate the application of our results is also a direction of our future work.

Acknowledgements. This paper is funded in part by the National Natural Science Foundation of China (62032019) and the Capacity Development Grant of Southwest University (SWU116007).

Disclosure of Interests. The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Akutsu, T., Hayashida, M., Ching, W.K., Ng, M.K.: Control of Boolean networks: hardness results and algorithms for tree structured networks. *J. Theor. Biol.* **244**(4), 670–679 (2007). <https://doi.org/10.1016/j.jtbi.2006.09.023>
2. Biane, C., Delaplace, F.: Abduction based drug target discovery using boolean control network. In: Feret, J., Koepl, H. (eds.) *CMSB 2017*. LNCS, vol. 10545, pp. 57–73. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67471-1_4

3. Cheng, D., Qi, H.: Controllability and observability of Boolean control networks. *Automatica* **45**(7), 1659–1667 (2009). <https://doi.org/10.1016/j.automatica.2009.03.006>
4. Cheng, D., Qi, H., Li, Z.: Analysis and control of Boolean networks: a semi-tensor product approach. *Acta Automatica Sinica* **37**, 529–540 (2011)
5. Cheng, D., Qi, H., Li, Z.: Identification of Boolean control networks. *Automatica* **47**(4), 702–710 (2011). <https://doi.org/10.1016/j.automatica.2011.01.083>
6. Cheng, D., Qi, H., Liu, T., Wang, Y.: A note on observability of Boolean control networks. *Syst. Control Lett.* **87**, 76–82 (2016). <https://doi.org/10.1016/j.sysconle.2015.11.004>
7. Fornasini, E., Valcher, M.E.: Observability, reconstructibility and state observers of Boolean control networks. *IEEE Trans. Autom. Control* **58**(6), 1390–1401 (2013). <https://doi.org/10.1109/TAC.2012.2231592>
8. Guo, Y.: Observability of Boolean control networks using parallel extension and set reachability. *IEEE Trans. Neural Netw. Learn. Syst.* **29**(12), 6402–6408 (2018). <https://doi.org/10.1109/TNNLS.2018.2826075>
9. Ideker, T., Galitski, T., Hood, L.: A new approach to decoding life: systems biology. *Ann. Rev. Genomics Hum. Genet.* **2**(1), 343–372 (2001). <https://doi.org/10.1146/annurev.genom.2.1.343>
10. Jacob, F., Monod, J.: Genetic regulatory mechanisms in the synthesis of proteins. *J. Mol. Biol.* **3**(3), 318–356 (1961). [https://doi.org/10.1016/S0022-2836\(61\)80072-7](https://doi.org/10.1016/S0022-2836(61)80072-7)
11. Kauffman, S.A.: Metabolic stability and epigenesis in randomly constructed genetic nets. *J. Theor. Biol.* **22**(3), 437–467 (1969). [https://doi.org/10.1016/0022-5193\(69\)90015-0](https://doi.org/10.1016/0022-5193(69)90015-0)
12. Kaufman, M., Andris, F., Leo, O.: A logical analysis of t cell activation and anergy. *Proc. Natl. Acad. Sci.* **96**(7), 3894–3899 (1999). <https://doi.org/10.1073/pnas.96.7.3894>
13. Klamt, S., Saez-Rodriguez, J., Lindquist, J.A., Simeoni, L., Gilles, E.D.: A methodology for the structural and functional analysis of signaling and regulatory networks. *BMC Bioinf.* **7**(1), 56 (2006). <https://doi.org/10.1186/1471-2105-7-56>
14. Li, F., Sun, J.: Observability analysis of Boolean control networks with impulsive effects. *IET Control Theory Appl.* **5**, 1609–1616 (2011). <https://doi.org/10.1049/iet-cta.2010.0558>
15. Mandon, H., Su, C., Haar, S., Pang, J., Paulevé, L.: Sequential reprogramming of boolean networks made practical. In: Bortolussi, L., Sanguinetti, G. (eds.) CMSB 2019. LNCS, vol. 11773, pp. 3–19. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31304-3_1
16. Su, C., Pang, J.: Target control of asynchronous Boolean networks. *IEEE/ACM Trans. Comput. Biol. Bioinf.* **20**(1), 707–719 (2023)
17. Su, C., Pang, J., Paul, S.: Towards optimal decomposition of Boolean networks. *IEEE/ACM Trans. Comput. Biol. Bioinf.* **18**(6), 2167–2176 (2021)
18. Su, C., Paul, S., Pang, J.: Controlling large Boolean networks with temporary and permanent perturbations. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) FM 2019. LNCS, vol. 11800, pp. 707–724. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30942-8_41
19. Thunberg, J., Ogren, P., Hu, X.: A Boolean control network approach to pursuit evasion problems in polygonal environments. In: IEEE International Conference on Robotics and Automation, pp. 4506–4511 (2011). <https://doi.org/10.1109/ICRA.2011.5979948>

20. Wang, B., Feng, J., Cheng, D.: On identification of Boolean control networks. *SIAM J. Control. Optim.* **60**(3), 1591–1612 (2022). <https://doi.org/10.1137/20M1373773>
21. Wu, G., Pang, J.: Single-experiment reconstructibility of Boolean control networks revisited. In: *Proceedings of the 20th International Conference on Informatics in Control, Automation and Robotics*, pp. 95–93. SCITEPRESS (2023)
22. Yuan, Q., Mizera, A., Pang, J., Qu, H.: A new decomposition-based method for detecting attractors in synchronous Boolean networks. *Sci. Comput. Program.* **180**, 18–35 (2019)
23. Zhang, K., Zhang, L.: Observability of Boolean control networks: a unified approach based on finite automata. *IEEE Trans. Autom. Control* **61**(9), 2733–2738 (2016). <https://doi.org/10.1109/TAC.2015.2501365>
24. Zhang, K., Zhang, L., Xie, L.: Detectability of nondeterministic finite-transition systems. In: *Discrete-Time and Discrete-Space Dynamical Systems. CCE*, pp. 165–175. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-25972-3_8
25. Zhang, K., Zhang, L., Xie, L.: Different types of discrete-time and discrete-space dynamical systems. In: *Discrete-Time and Discrete-Space Dynamical Systems. CCE*, pp. 35–56. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-25972-3_2
26. Zhang, K., Zhang, L., Xie, L.: Observability and detectability of large-scale boolean control networks. In: *Discrete-Time and Discrete-Space Dynamical Systems. CCE*, pp. 117–142. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-25972-3_6
27. Zhang, K., Zhang, L., Xie, L.: Observability of Boolean control networks. In: *Discrete-Time and Discrete-Space Dynamical Systems. CCE*, pp. 87–104. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-25972-3_4
28. Zhao, Y., Qi, H., Cheng, D.: Input-state incidence matrix of Boolean control networks and its applications. *Syst. Control Lett.* **59**(12), 767–774 (2010). <https://doi.org/10.1016/j.sysconle.2010.09.002>
29. Zhong, D., Li, Y., Lu, J.: Feedback stabilization of Boolean control networks with missing data. *IEEE Trans. Neural Netw. Learn. Syst.* **34**, 7784–7795 (2023). <https://doi.org/10.1109/TNNLS.2022.3146262>
30. Zhu, S., Lu, J., Lin, L., Liu, Y.: Minimum-time and minimum-triggering observability of stochastic Boolean networks. *IEEE Trans. Autom. Control* **67**(3), 1558–1565 (2022). <https://doi.org/10.1109/TAC.2021.3069739>
31. Zhu, S., Lu, J., Zhong, J., Liu, Y., Cao, J.: On the sensors construction of large Boolean networks via pinning observability. *IEEE Trans. Autom. Control* (2021). <https://doi.org/10.1109/TAC.2021.3110165>



Formalizing Potential Flows Using the HOL Light Theorem Prover

Elif Deniz^(✉) and Sofiène Tahar

Department of Electrical and Computer Engineering, Concordia University,
Montreal, QC, Canada
{e_deniz,tahar}@ece.concordia.ca

Abstract. Potential flow is a theoretical model that describes the movement of a fluid, e.g., water or air in situations where viscosity and turbulence are assumed to be negligible. This type of flow is often used as an idealized model to describe the behavior of fluids in specific contexts, such as in fluid dynamics and aerodynamics. In this paper, we present a higher-order logic formalization of potential flows that are governed by the Laplace equation. We focus on formally modeling fundamental flows such as the uniform, source/sink, doublet, and vortex flows in the HOL Light theorem prover. We then prove the validity of these exact potential flow solutions of the Laplace equation. Moreover, we present the formal verification of the linearity of the Laplace operator, which is essential to apply the superposition principle. To demonstrate the practical effectiveness of our formalization, we formally verify several applications such as the Rankine oval, flow past a circular cylinder and flow past a rotating circular cylinder, each of which involves combining these standard flows using the superposition principle to model more complex fluid dynamics.

Keyword: Potential Flows, Partial Differential Equations, Laplace Equation, Higher-Order Logic, Theorem Proving, HOL Light

1 Introduction

Potential flow theory [14] is a key concept in the discipline of fluid dynamics. It uses harmonic functions to study a wide range of fluid-related phenomena within the theoretical framework of this field of study. Potential flow describes the velocity field as the gradient of a scalar function known as the velocity potential. Moreover, it characterizes the flow as irrotational and incompressible and provides valuable insights into fluid dynamics. This idealization is in close approximation to real-world scenarios of practical importance. For instance, in aerodynamics, this theory has played a pivotal role in developing analytical models to understand airflow around airfoils, wings, and related aerodynamic surfaces, which in turn facilitate the prediction of crucial aerodynamic forces such as lifts [13].

The foundation of addressing aerodynamic problems lies in the equations that govern the flow. While fluid motion is governed by the Navier-Stokes (NS)

equations [18], which is a vector equation that includes three different scalar equations along with the conservation of the mass equation [19], their nonlinear nature renders them challenging to solve [14]. Consequently, the Laplace equation, which is a prevalent class of partial differential equations [17] emerges as a preferred alternative, providing an exact representation of incompressible, inviscid and irrotational flows. Unlike the NS equations, the use of the Laplace equation is much easier than using fully viscous NS equations. This equation forms the basis of potential flow theory, where both the stream function and velocity potential, as algebraic functions satisfying the Laplace equation, can be combined to construct flow fields. Moreover, the superposition of basic potential flow solutions is a crucial step in the analysis of aerodynamic configurations. This method leverages the linearity of the Laplace equation, enabling for the construction of models that represent intricate scenarios by combining simpler flow elements [16].

Due to the fundamental importance of the Laplace equation in physics, applied mathematics, and engineering, numerous well-established analytical and numerical techniques exist for solving this equation, especially in the field of aerodynamics. These techniques are also useful in developing advanced computational methods for determining potential flows around the complex three-dimensional geometries common in modern aircraft design [13]. For instance, the method of images [9] are applied to model potential flows around airfoils and wings, where a combination of real and image sources helps satisfy the no-flow boundary conditions on solid surfaces. On the other hand, numerical techniques such as the panel methods [3] are computational models that simplify the assumptions concerning the aerodynamic principles and characteristics of airflow over an aircraft. Despite the prevalence of traditional techniques in analyzing aerodynamic problems, there exists a notable concern regarding their accuracy. For instance, paper-and-pencil methods carry a risk of human errors. It is possible that a mathematical result may be misapplied when using a manual method, as it is not possible to guarantee that all required assumptions are valid. In regard to simulation tools, the accuracy of simulation results depends on various factors, including the precision of numerical techniques, and computational issues may arise, especially in the context of large models.

In contrast, formal verification employs computer-based techniques for the mathematical modeling, analysis, and verification of abstract and physical systems. A prominent technique in formal verification is higher-order logic (HOL) theorem proving [11], which is an interactive approach that involves human-machine collaboration for the development of correct proofs. Its expressive capabilities are sufficient for the description of the majority of classical mathematical theories, including differentiation, integration, higher transcendental functions, and topological spaces. Given the fundamental role of potential flow theory in the early stages of aircraft design, where it is used to predict the behavior of airflow around wings, the safety-critical nature of potential flow applications becomes evident. Therefore, it is imperative to employ robust verification tools that can ensure the accuracy and reliability of these theoretical models.

In this paper, we propose to use higher-order logic theorem proving for the formalization of standard potential flows that are governed by the Laplace equation. We also provide the formal verification of these exact potential flow solutions for the Laplace equation, along with their applications in aerodynamics. While there exist some formalization work of other types of partial differential equations, such as the wave equation [4], the heat equation [7] and the telegrapher's equations [8], to the best of our knowledge, there is no formalization of the Laplace equation in the literature. Therefore, the formal analysis of potential flows governed by the Laplace equation using HOL theorem proving is the first of its kind, which could be very useful for safety-critical applications.

The rest of the paper is organized as follows: Sect. 2 describes some preliminary details of the potential flow theory and the HOL Light theorem prover that are necessary for understanding the rest of the paper. We present the formalization of standard potential flows in Sect. 3. In Sect. 4, we provide the formal verification of the validity of the exact potential flow solutions for the Laplace equation. Sect. 5 provides the formal verification of the linearity of the Laplace operator as well as the verification of more complicated flows that are constructed by combining the standard potential flows. Finally, Sect. 6 concludes the paper.

2 Preliminaries

In this section, we briefly describe the HOL Light theorem prover as well as some of the associated functions and symbols that are necessary for understanding the rest of the paper. We also provide some background knowledge about potential flow theory.

2.1 HOL Light Theorem Prover

Interactive theorem proving is a collaborative process between a machine and a human user, where they work together interactively to generate a formal proof. The use of theorem proving systems is common in the verification of both software and hardware as well as in pure mathematics. For instance, a verification engineer can manually build a logical model of the system and subsequently verify the desired properties while providing guidance to the theorem proving tool. Similarly, a mathematician can use theorem provers in the verification of standard pure mathematical contexts. HOL Light [12], developed by Harrison, is one of the theorem provers in the HOL family [11], characterized by its small logical kernel. In HOL Light, the process of proving a theorem begins with the user entering the theorem's statement as the goal in a new proof. The proofs in HOL Light rely on tactics that break down complex goals into more straightforward subgoals. Furthermore, HOL Light provides a variety of automated proof procedures and proof assistants to assist users in guiding and completing their proofs. In addition, users have the flexibility to craft and implement their own personalized automation methods.

Table 1 provides the mathematical interpretations of some of the HOL Light symbols and functions used in this paper.

Table 1. HOL Light Symbols

HOL Light Symbols	Standard Symbols	Description
<code>&a</code>	$\mathbb{N} \rightarrow \mathbb{R}$	Type casting from natural numbers to reals
<code>&num</code>	$\{1, 2..\}$	Positive integers data type
<code>$\lambda x. t$</code>	$\lambda x. t$	Function that maps x to $t(x)$
<code>real</code>	\mathbb{R}	Real data type
<code>@f</code>	Hilbert choice operator	Returns f if it exists
<code>atreal x</code>	Real net	At real variable x
<code>--x</code>	$-x$	Unary negation of x
<code>a / b</code>	$\frac{a}{b}$	Division (a and b should have same type)
<code>a pow b</code>	a^b	Real or complex power

2.2 Brief Review of Potential Flow Theory

Potential flow can be defined as steady, incompressible and irrotational flow. A condition that is necessary and sufficient to identify a flow as irrotational:

$$\vec{\nabla} \times \vec{V} = 0 \tag{1}$$

This indicates that the velocity field \mathbf{V} is a conservative vector field denoted by the gradient of a scalar velocity potential function (ϕ):

$$\vec{V} = \vec{\nabla} \phi \tag{2}$$

If the velocity potential is known, then the velocity at any point can be determined using

$$u = \frac{\partial \phi}{\partial x}, \quad v = \frac{\partial \phi}{\partial y} \tag{3}$$

The irrotationality condition for two-dimensional flows vorticity is given by:

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \xi \tag{4}$$

Here, $\xi = 0$ since the flow is irrotational.

Similarly, in the case of an incompressible flow, it follows from the continuity equation that:

$$\vec{\nabla} \cdot \vec{V} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{5}$$

The two-dimensional continuous flow is described by the stream function (for incompressible flow) ψ , which determines the velocity at any point as:

$$u = \frac{\partial\psi}{\partial y}, \quad v = -\frac{\partial\psi}{\partial x} \quad (6)$$

Substituting Eqs. (3) and (6) into Eqs. (5) and (4), respectively, yields the conditions for continuous irrotational flow:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 = \frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} \quad (7)$$

which is the Laplace equation for the stream function and the velocity potential in Cartesian coordinates [13]. The Laplace equation can also be written in polar coordinates as:

$$\frac{\partial^2\psi}{\partial r^2} + \frac{1}{r} \frac{\partial\psi}{\partial r} + \frac{1}{r^2} \frac{\partial^2\psi}{\partial\theta^2} = 0 \quad (8)$$

Both the velocity potential (ϕ) and the stream function (ψ) are employed to describe the flow field in fluid dynamics and they satisfy the Laplace equation. There are notable similarities and differences between the stream function and the velocity potential. For instance, while the stream function can be employed to describe both rotational and irrotational flows, the velocity potential is only defined for irrotational flow. On the other hand, the velocity potential is applicable to three-dimensional flows, whereas the stream function has only been defined for two-dimensional flows.

There are several techniques available to determine both the velocity potential (ϕ) and the stream function (ψ). For instance, common numerical and analytical techniques such as Finite Element Method (FEM) [5] and separation of variables [10], respectively are frequently used to solve the Laplace equation with the appropriate boundary conditions. Another popular technique is to find some simple functions that satisfy the Laplace equation and to model the flow around the body of interest, which is possible due to the linearity of the Laplace equation. The focus of this paper will be this latter method, which is the most widely used procedure for potential flows. In the next section, we will present the formalization of these basic flows.

3 Formalizing Standard Potential Flow Solutions

In this section, we present some basic functions which satisfy the Laplace equation. Any function that satisfies this equation describes a potential flow. It is noteworthy that in this work, we are interested in employing exact potential flow solutions to formally validate them for the Laplace equation. Furthermore, our objective is to use these elementary flows as building blocks to construct a desired flow field, rather than deriving them.

3.1 Uniform Flow

The most basic type of flow is a uniform steady flow as shown in Fig. 1. A uniform flow directed in the positive x -direction has the velocity components $u = U$ and $v = 0$ everywhere. This type of flow is irrotational and therefore possesses a velocity potential ϕ , which can be shown as follows:

$$\phi = Ux \tag{9}$$

Additionally, the stream function can be expressed as:

$$\psi = Uy \tag{10}$$

The formal representations of a uniform flow for the stream function and the velocity potential are given as follows:

Definition 1. *Uniform Flow*

$\vdash_{def} \forall \mathbf{U} \mathbf{y}. \text{stream.uniform } \mathbf{U} \mathbf{y} = \mathbf{U} * \mathbf{y}$
 $\vdash_{def} \forall \mathbf{U} \mathbf{y}. \text{velocity.uniform } \mathbf{U} \mathbf{x} = \mathbf{U} * \mathbf{x}$

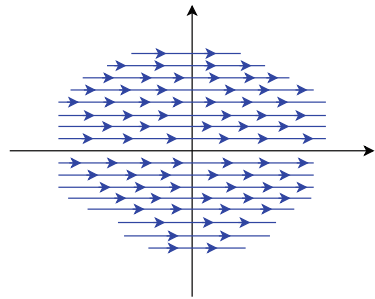


Fig. 1. Uniform Flow

3.2 Source/Sink Flow

In two-dimensional fluid dynamics, a source is defined as a point where fluid propagates radially outward, while a sink represents a point of negative source characterized by inward radial fluid movement as illustrated in Fig. 2(a) and 2(b), respectively.

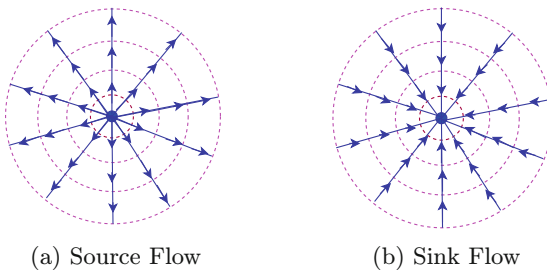


Fig. 2. Source/Sink Flow

The exact potential flow solutions centered at point (x_0, y_0) for the stream function and the velocity potential are mathematically expressed as [13]:

$$\psi(x, y) = \frac{m}{2\pi} \tan^{-1} \left(\frac{y - y_0}{x - x_0} \right) \tag{11}$$

$$\phi(x, y) = \frac{m}{4\pi} \ln((x - x_0)^2 + (y - y_0)^2) \tag{12}$$

Here, m denotes the strength of the source. A positive m ($m > 0$) denotes a source flow, whereas a negative m ($m < 0$) indicates a sink flow.

Now, we formalize the above equations, i.e., Eqs. (11) and (12) in HOL Light as follows:

Definition 2. *Source Flow for the Stream Function*

$$\vdash_{def} \forall m \ x \ y \ x0 \ y0. \\ \text{stream_source } m \ x \ y \ x0 \ y0 = \\ m / (\&2 * \text{pi}) * \text{atn} ((y - y0) / (x - x0))$$

Definition 3. *Source Flow for the Velocity Potential*

$$\vdash_{def} \forall m \ x \ y \ x0 \ y0. \\ \text{velocity_source } m \ x \ y \ x0 \ y0 = \\ m / (\&4 * \text{pi}) * \text{log} ((x - x0) \text{ pow } 2 + (y - y0) \text{ pow } 2)$$

Here, `atn` and `log` indicate the inverse of the tangent function and the natural logarithm, respectively.

In the next subsections, we will use the polar coordinates r and θ to describe the doublet and vortex flows. Note that uniform and source/sink flows can be similarly represented using polar coordinates, utilizing the relationships $x = r \cos \theta$, $y = r \sin \theta$. These transformations are particularly useful for practical examples.

3.3 Doublet Flow

As depicted in Fig. 3, the doublet is a special flow pattern that arises when a source and a sink of equal strength are constrained to have a constant ratio of strength to distance (κ), as the distance approaches zero.

The resulting solutions for the stream function and the velocity potential are as follows:

$$\psi(r, \theta) = -\frac{\kappa}{2\pi r} \sin\theta \tag{13}$$

$$\phi(r, \theta) = \frac{\kappa}{2\pi r} \cos\theta \tag{14}$$

The next step is to formalize the above equations (Eqs. (13) and (14)) in HOL Light:

Definition 4. *Doublet Flow for the Stream Function*

$$\vdash_{def} \forall K \ \text{theta} \ r. \\ \text{stream_doublet } K \ \text{theta} \ r = \\ --(K / (\&2 * \text{pi} * r)) * \text{sin} (\text{theta})$$

Definition 5. *Doublet Flow for the Velocity Potential*

$$\vdash_{def} \forall K \ \text{theta} \ r. \\ \text{velocity_doublet } K \ \text{theta} \ r = \\ (K / (\&2 * \text{pi} * r)) * \text{cos} (\text{theta})$$

where `stream_doublet` and `velocity_doublet` accept the strength K , the radius r and the angle theta and return the corresponding functions.

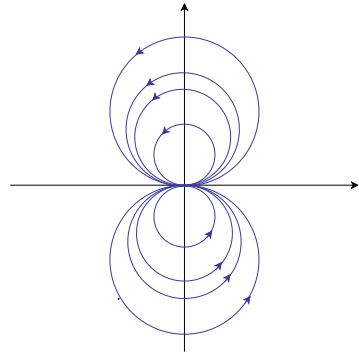


Fig. 3. Doublet Flow

3.4 Vortex Flow

A two-dimensional, steady flow that circulates about a point is known as a line vortex. In this type of flow, the streamlines form concentric circles around a specific point as shown in Fig. 4. It is important to note that the irrotational nature of the flow is not contradicted by the potential vortex formulation.

Fluid elements travel in a circular path around the vortex centre without rotating about their axes, thus meeting the condition of irrotational flow. The exact potential flow solution centered at the origin is mathematically expressed as:

$$\psi(r, \theta) = \frac{\Gamma}{2\pi} \ln(r) \tag{15}$$

$$\phi(r, \theta) = -\frac{\Gamma}{2\pi} \theta \tag{16}$$

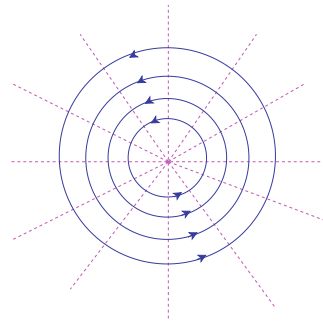


Fig. 4. Vortex Flow

where Γ represents the circulation, which is often positive when moving counter-clockwise.

Next, we formalize the vortex flow for the stream function and the velocity potential, i.e., Eqs. (15) and (16) as:

Definition 6. *Vortex Flow for the Stream Function*

$$\vdash_{def} \forall \gamma \in \mathbb{R}. \text{stream_vortex } \gamma \text{ } r = \gamma / (2 * \pi) * \log(r)$$

Definition 7. *Vortex Flow for the Velocity Potential*

$$\vdash_{def} \forall \gamma \in \mathbb{R}. \text{velocity_vortex } \gamma \text{ } \theta = -\gamma / (2 * \pi) * \theta$$

Table 2 summarizes the potential flows that are presented in this section.

Table 2. Standard Flows Overview

Flow Type	Stream Function	Velocity Potential
Uniform flow in the x -direction	$\psi(x, y) = Uy$	$\psi(x, y) = Ux$
Source/Sink	$\psi(x, y) = \frac{m}{2\pi} \tan^{-1} \left(\frac{y - y_0}{x - x_0} \right)$	$\phi(x, y) = \frac{m}{4\pi} \ln((x - x_0)^2 + (y - y_0)^2)$
Doublet	$\psi(r, \theta) = -\frac{\kappa}{2\pi r} \sin\theta$	$\phi(r, \theta) = \frac{\kappa}{2\pi r} \cos\theta$
Vortex	$\psi(r, \theta) = \frac{\Gamma}{2\pi} \ln(r)$	$\phi(r, \theta) = -\frac{\Gamma}{2\pi} \theta$

4 Formal Verification of the Laplace Equation's Solutions

In this section, we present the formal verification of the exact potential flow solutions of the Laplace equation. The purpose of this verification is to ensure the correctness of analytical solutions and then establish their foundational role in describing fluid behavior and facilitating engineering applications.

For this verification, our first step is to formalize the Laplace equation in both Cartesian and polar coordinates in the HOL Light as follows:

Definition 8. *The Laplace Equation in Cartesian Coordinates*

$$\vdash_{def} \text{laplace_equation } \psi(x,y) \Leftrightarrow \text{laplace_operator } \psi(x,y) = \&0$$

where `laplace_equation` accepts the real function $\psi: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, the space variables $x:\mathbb{R}$ and $y:\mathbb{R}$ and returns the corresponding Laplace equation. The function Laplace operator is formalized as:

Definition 9. *Laplace Operator* $\vdash_{def} \forall \psi \ x \ y.$

$$\begin{aligned} \text{laplace_operator } \psi(x,y) = \\ \text{higher_real_derivative } 2 \ (\lambda x. \psi(x,y)) \ x + \\ \text{higher_real_derivative } 2 \ (\lambda y. \psi(x,y)) \ y \end{aligned}$$

Here, `higher_real_derivative` represents the n^{th} -order real derivative of a function.

The formal representation of the Laplace equation in polar coordinates, i.e., Eq. (8) is formalized as follows:

Definition 10. *The Laplace Equation in Polar Coordinates*

$$\begin{aligned} \vdash_{def} \forall \psi \ r \ \theta. \text{laplace_in_polar } \psi \ r \ \theta = \\ \text{higher_real_derivative } 2 \ (\lambda r. \psi(r,\theta)) \ r + \\ \&1/r * \text{higher_real_derivative } (\lambda r. \psi(r,\theta)) \ r + \\ \&1/(r \text{ pow } 2) * \text{higher_real_derivative } (\lambda \theta. \psi(r,\theta)) \ \theta = \&0 \end{aligned}$$

where the HOL Light function `laplace_in_polar` mainly accepts the function ψ of type $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, the radial distance r and the angle θ and returns the corresponding equation. We can also formalize the Laplace equation for the velocity potential in a similar manner. With the formal definitions outlined previously, an important step is to verify that these potential flow solutions satisfy the Laplace equation. In other words, this is the main condition for potential flows to be valid, which is fundamental for understanding fluid behavior in various contexts. We start with the verification of the source flow for the stream function, i.e., Eq. (11) in HOL Light as follows:

Theorem 1. *Verification of the Source Flow for the Stream Function*

$$\vdash_{thm} \forall m \ x0 \ y0 \ \psi.$$

$$[A1] \ (\forall x. \ x \neq x0) \wedge [A2] \ (\forall y. \ y \neq y0) \wedge$$

$$[A3] \ (\forall x \ y. \ \psi(x,y) = \text{stream_source } m \ x \ y \ x0 \ y0)$$

$$\Rightarrow \text{laplace_equation } \psi \ x \ y$$

Assumptions A1 and A2 ensure that the points in a Cartesian coordinate system are different from each other. Assumption A3 provides the solution of the Laplace equation for source flow, i.e., Eq. (11). The proof of the above theorem is mainly based on the real differentiation of the source flow solution with respect to the parameters x and y .

Our next step is to formally verify the doublet flow (Eq. (13)) as the following HOL Light theorem:

Theorem 2. *Verification of the Doublet Flow for the Stream Function*

$\vdash_{thm} \forall K \ u.$

[A1] $(\lambda r. \&0 < r) \wedge$

[A2] $(\forall r \ \theta. \text{psi}(r, \theta) = \text{stream_doublet } K \ \theta \ r)$

$\Rightarrow \text{laplace_in_polar } \text{psi} \ r \ \theta$

Assumption A1 ensures that the radial distance is greater than zero. Assumption A2 provides the solution of the Laplace equation in polar coordinates (Eq. (8)) for doublet flow (Eq. (13)). The verification of Theorem 2 is mainly based on the properties of real derivative [1] and some real arithmetic reasoning.

Finally, the vortex flow, i.e., Eq. (15) is verified as the following theorem:

Theorem 3. *Verification of the Vortex Flow for the Stream Function*

$\vdash_{thm} \forall \gamma \ u.$

[A1] $(\lambda r. \&0 < r) \wedge$

[A2] $(\forall r \ \theta. \text{psi}(r, \theta) = \text{stream_vortex } \gamma \ u \ r \ \theta)$

$\Rightarrow \text{laplace_in_polar } \text{psi} \ r \ \theta$

Assumption A1 is the same as that of Theorem 2. A2 provides the vortex flow solution for the stream function, i.e., Eq. 15. The conclusion of Theorem 3 provides that the vortex flow solution satisfies the Laplace equation. The proof of Theorem 3 is primarily based on the real differentiation of the vortex flow solution with respect to the parameters r and θ . In this section, we only presented the theorems for the stream function for the sake of brevity. The verification of the velocity potential function is done in a similar way. Details about verification of the rest of the theorems can be found in our proof script [6].

In the next section, we use these formally verified solutions to build more complicated flows which are widely applied in the analysis of flow patterns around an airfoil [15].

5 Applications of Standard Flows

The Laplace equation is a second-order, linear, elliptic partial differential equation. Thanks to the linearity of the Laplace equation, more complicated flow fields can be constructed from the superposition of basic solutions. If ψ_1 and ψ_2 are the solutions (stream functions) of the Laplace's equation and then their

linear combination $\psi_1 + \psi_2$ will also be a solution for a two-dimensional incompressible and irrotational flow. This unique feature makes this equation a powerful tool to analyze fluid flow problems. The ability to obtain new flow patterns by superimposing known flows is fundamental to wing theory, as it provides simple solutions to complex problems [2].

Our first step is to formally verify the linearity of the Laplace operator due to its importance for the superposition principle.

Theorem 4. *Linearity of Laplace Operator*

$\vdash_{thm} \forall \text{psi phi a b.}$

```
[A1] (∀x. (λx. psi(x,y)) real_differentiable atreal x) ∧
[A2] (∀x. (λx. phi(x,y)) real_differentiable atreal x) ∧
[A3] (∀y. (λy. psi(x,y)) real_differentiable atreal y) ∧
[A4] (∀y. (λy. phi(x,y)) real_differentiable atreal y) ∧
[A5] (∀x. (λx. real_derivative (λx. psi(x,y)) x)
      real_differentiable atreal x) ∧
[A6] (∀x. (λx. real_derivative (λx. phi(x,y)) x)
      real_differentiable atreal x)
[A7] (∀y. (λy. real_derivative (λx. psi(x,y)) y)
      real_differentiable atreal y)
[A8] (∀y. (λy. real_derivative (λy. phi(x,y)) y)
      real_differentiable atreal y)
⇒ laplace_operator (λ(x,y). a * psi(x,y) + b * phi(x,y)) (x,y) =
   a * laplace_operator (λ(x,y). psi(x,y)) (x,y) +
   b * laplace_operator (λ(x,y). phi(x,y)) (x,y)
```

Assumptions A1 and A2 ensure that the real-valued functions `psi` and `phi` are differentiable at `x`, respectively. Assumptions A3 and A4 assert the differentiability of the functions `psi` and `phi` at `y`, respectively. Additionally, Assumptions A5 and A6 provide the differentiability conditions for the derivatives of the functions `psi` and `phi` at `x`, respectively. Similarly, Assumptions A7 and A8 guarantee the differentiability conditions for the derivatives of the functions `psi` and `phi` at `y`, respectively. The proof of the above theorem mainly relies on the properties of derivatives and the differentiability of real-valued functions.

5.1 The Rankine Oval

By combining the exact solutions for uniform and source/sink flows, we can construct a flow field around an oval-shaped object. The resultant configuration is known as the Rankine oval. We start by analyzing the flow pattern around a source and a sink. The source and sink are placed along the x -axis, separated by a distance of $2a$, as depicted in Fig. 5(a). The origin is situated equidistantly between them. We now superimpose the uniform, source and sink flows, all positioned in the x -direction, with a line source located at $(-a, 0)$ and a line sink of equal and opposite strength located at $(+a, 0)$, as depicted in Fig. 5(b). Assume the strengths of these source and the sink are $+m$ and $-m$, respectively.

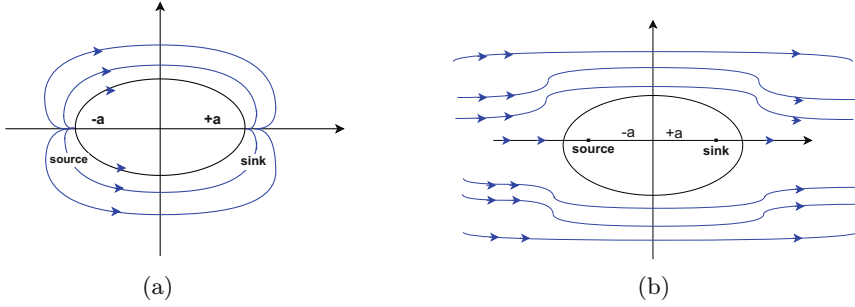


Fig. 5. Source/Sink Flow

The overall stream function (ψ) and velocity potential (ϕ) for this combination of flows are expressed as:

$$\psi = \psi_{uniform} + \psi_{source} + \psi_{sink} \tag{17}$$

$$\phi = \phi_{uniform} + \phi_{source} + \phi_{sink} \tag{18}$$

Mathematically, they are represented by the combination of Eqs. (9), (10), (11) and (12) for the stream function and the velocity potential as:

$$\psi(x, y) = -Uy + \frac{m}{2\pi} \left[\arctan\left(\frac{y}{x+a}\right) - \arctan\left(\frac{y}{x-a}\right) \right] \tag{19}$$

$$\phi(x, y) = Ux + \frac{m}{4\pi} \ln\left(\frac{(x+a)^2 + y^2}{(x-a)^2 + y^2}\right) \tag{20}$$

Next, we formally verify these combined flows for the stream function as the following HOL Light theorem:

Theorem 5. *Verification of the Rankine Oval for the Stream Function*

$\vdash_{thm} \forall U \ m \ a \ \text{psi} \ x0 \ x1 \ y0 \ y1.$

[A1] $(\forall x. x \neq a) \wedge$ [A2] $(\forall x. x \neq -a) \wedge$ [A3] $x0 = -a \wedge$

[A4] $x1 = a \wedge$ [A5] $y0 = \&0 \wedge$ [A6] $y1 = \&0 \wedge$

[A7] $(\forall x \ y. \text{psi}(x,y) = \text{sum } (0..2) (\lambda n. \text{EL } n \ [\text{--stream.uniform } U \ y;$
 $\text{stream.source } m \ x \ y \ x0 \ y0; \text{stream.sink } m \ x \ y \ x1 \ y1]))$

$\Rightarrow \text{laplace_equation } \text{psi} \ x \ y$

Assumptions A1 and A2 guarantee that the validity of our expression by specifying that x must be different from a and $-a$, respectively. Assumptions A3 and A4 provide the distance from the origin. Assumptions A5 and A6 assert that the points $y0$ and $y1$ are equal to zero since the flows are oriented in towards the x -direction. Assumption A7 provides the combined solutions for the stream function, i.e., Eq. (19). Here, the function $\text{EL } n \ l$ extracts the n^{th} element from a list l . The verification of Theorem 5 is mainly based on the properties of real derivatives, some real arithmetic reasoning and the following HOL Light lemma:

Lemma 1. *Superposition of the Solutions*

$\vdash_{lem} \forall U m x y x0 x1 y0 y1.$
 $sum (0..2) (\lambda n. EL n [--stream.uniform U y; stream.source m x y x0 y0;$
 $stream.sink m x y x1 y1]) = --stream.uniform U y + stream.source m x y x0 y0$
 $+ stream.sink m x y x1 y1$

The above lemma states that the summation of the list equals to the linear combination of uniform, source and sink flows.

5.2 Potential Flow Past a Circular Cylinder

As shown in Fig. 6, we can build a potential flow solution for the flow around a circular cylinder using the superposition of a uniform (Fig. 6(a)) and a doublet flow (Fig. 6(b)) in the x -direction. This combination produces a non-lifting flow over the cylinder, as represented in Fig. 6(c). The resulting stream function and velocity potential for this particular combination of potential flows can be given as:

$$\psi = \psi_{uniform} + \psi_{doublet} \tag{21}$$

$$\phi = \phi_{uniform} + \psi_{doublet} \tag{22}$$

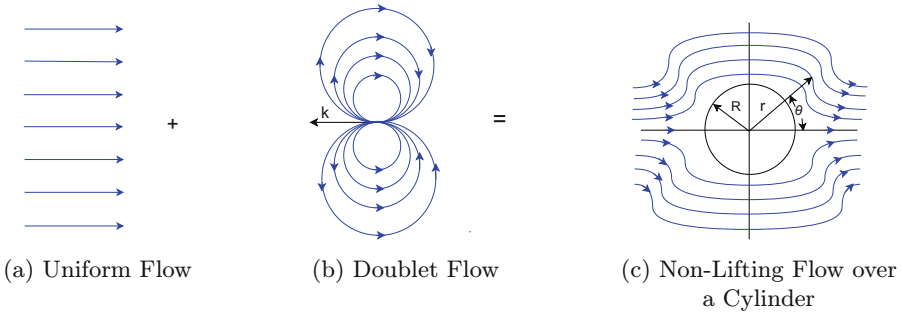


Fig. 6. Potential Flow Past a Circular Cylinder [14]

We can mathematically express this combination by adding the solutions for uniform and doublet flows, i.e., Eqs. (9), (10), (13) and (14). It is known that $y = r \sin\theta$ in polar coordinates.

$$\psi(r, \theta) = U \left(r + \frac{\kappa}{2\pi r} \right) \sin\theta \tag{23}$$

$$\phi(r, \theta) = U \left(r - \frac{\kappa}{2\pi r} \right) \cos\theta \tag{24}$$

Next, we formally verify Eq. (23) in HOL Light as follows:

Theorem 6. *Verification of Potential Flow Past a Circular Cylinder*

$\vdash_{thm} \forall U K y \text{ psi.}$

[A1] $(\forall r. \&0 < r) \wedge$ [A2] $(\forall r \text{ theta. } y = r * \sin(\text{theta})) \wedge$
 [A3] $(\forall r \text{ theta. } \text{psi}(r, \text{theta}) = \text{sum } (0..1) (\forall n. \text{EL } n [\text{stream_uniform } U \text{ y};$
 $\text{stream_doublet } K \text{ theta } r]))$
 $\Rightarrow \text{laplace_in_polar } \text{psi } r \text{ theta}$

Assumption A1 ensures that the radial distance is greater than zero, while Assumption A2 indicates that $y = r * \sin(\text{theta})$ in polar coordinates. Assumption A3 provides the superposition of the uniform and doublet flow solutions for the stream function, as shown in Eq. (23). Similar to Theorem 5, we proved a lemma regarding superposition of the solution as well as proving the real derivatives of the solution in order to formally verify this theorem.

5.3 Potential Flow Past a Rotating Circular Cylinder

Figure 7(c) illustrates a flow around a rotating circular cylinder. This flow can be constructed by combining a uniform flow and a doublet flow, as depicted in Fig. 7(a), along with a vortex flow, as shown in Fig. 7(b). In this context, the stream function and the velocity potential for this combination of potential flows can, respectively, be given as:

$$\psi = \psi_{uniform} + \psi_{doublet} + \psi_{vortex} \tag{25}$$

$$\phi = \phi_{uniform} + \phi_{doublet} + \phi_{vortex} \tag{26}$$

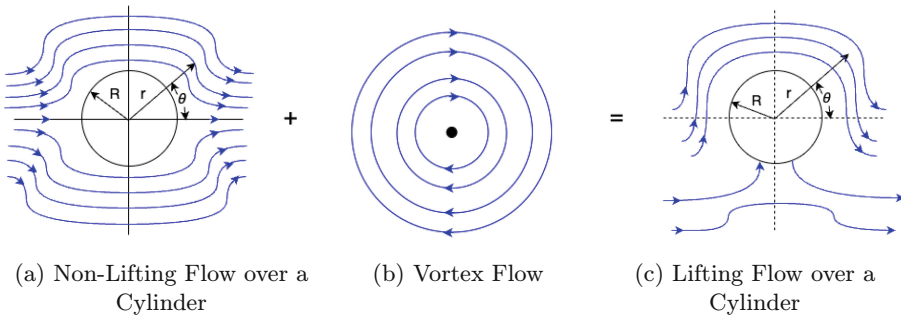


Fig. 7. Potential Flow Past a Rotating Circular Cylinder [14]

It is important to note that combining a uniform flow and a doublet flow effectively models the flow around a non-rotating circular cylinder, as given by Eqs. (23) and (24). Therefore, we can write the final mathematical expression of these flows for the stream function and the velocity potential by adding the solutions, i.e., Eqs. (15), (16), (23) and (24) as:

$$\psi(r, \theta) = U \left(r + \frac{\kappa}{2\pi r} \right) \sin\theta + \frac{\Gamma}{2\pi} \ln(r) \tag{27}$$

$$\phi(r, \theta) = U \left(r - \frac{\kappa}{2\pi r} \right) \cos\theta + -\frac{\Gamma}{2\pi} \theta \tag{28}$$

The above equations can be alternatively written as:

$$\psi(r, \theta) = U r \sin\theta \left(1 - \frac{R^2}{r^2} \right) + \frac{\Gamma}{2\pi} \ln r \tag{29}$$

$$\phi(r, \theta) = U r \cos\theta \left(1 - \frac{R^2}{r^2} \right) + \frac{\Gamma}{2\pi} \theta \tag{30}$$

where $R^2 = \frac{m}{2\pi U}$ and m is the strength of the doublet.

Finally, we formally verify Eq. (27) as the following HOL Light theorem:

Theorem 7. *Verification of Potential Flow Past a Rotating Circular Cylinder*
 $\vdash_{thm} \forall U \ K \ y \ \text{gamma} \ \text{psi}.$

```
[A1] (∀r. &0 < r) ∧ [A2] (∀r theta. y = r * sin(theta)) ∧
[A3] (∀r theta. psi(r,theta) = sum (0..2) (∀n. EL n [stream_uniform U y;
stream_doublet K theta r; stream_vortex gamma theta r]))
⇒ laplace_in_polar psi r theta
```

Assumptions A1-A2 are the same as those of Theorem 6. Assumption A3 provides the combination of the uniform, doublet and vortex flow solutions for the stream function, i.e., Eq. (27). The verification of Theorem 7 is similar to that of Theorem 6. We also conducted a formal verification of the combination of these standard flows for the velocity potential. Further details on this latter formalization can be found in our proof script [6].

5.4 Discussion

A notable aspect of the work presented in this paper is the development of the first formalization of potential flows which has wide applications in aerodynamics, particularly in airfoil theory. A key aspect of our work is the incorporation of theorem proving into a domain typically prevalent in numerical techniques. This approach allows for the identification of logical errors and inconsistencies in models that may not be evident in simulation results, ultimately helping to prevent potential flaws during the design process. One of the main challenges of this work is its interdisciplinary nature, as it requires a deep understanding of aerodynamic principles, the integration of mathematics, and the meticulous process of interactive theorem proving. Another significant challenge is verifying exact analytical solutions governed by the Laplace equation. The proof process must establish the real derivatives of these solutions and their linear combinations. While traditional paper-and-pencil proofs can overlook trivial details, theorem proving demands a substantial amount of time due to the undecidable nature of

higher-order logic and requires every detail to be meticulously provided to the computer. One of the benefits of this work is that it addresses these challenges by formalizing the core concepts of potential flow theory, allowing available results to be built upon to minimize user interaction. Additionally, all of the verified theorems and lemmas are general, opening the door to future expansions. Given the limited number of engineers and physicists with expertise in formal methods, we believe that our work can be a significant step towards bridging the gap between theorem proving and the aerospace engineering communities, thereby enhancing its applicability in industrial settings.

6 Conclusion

In this paper, we conducted the formal specification and verification of standard potential flows solutions which satisfy the Laplace equation using higher-order logic theorem proving. We first formalized four fundamental potential flows, namely, the uniform, source/sink, doublet and vortex flows. Moreover, we formally modeled the Laplace equation in both Cartesian and polar coordinates. Furthermore, we formally verified the linearity of the Laplace operator since it is a very powerful tool to create more complicated flow fields. We then constructed the formal proof for the exact potential flow solutions of the Laplace equation. Finally, in order to demonstrate the applicability of our formalization work, we formally analyzed several practical applications, including the Rankine oval, potential flow past a circular cylinder and potential flow past a rotating circular cylinder. For the future work, we plan to extend our formalization for other complex-valued potential flows in order to analyze more complicated problems in aerodynamics.

References

1. HOL Light Real Calculus (2024). <https://github.com/jrh13/hol-light/blob/master/Multivariate/realanalysis.ml>
2. Abbott, I.H., Von Doenhoff, A.E.: Theory of Wing Sections: Including a Summary of Airfoil Data. Courier Corporation (2012)
3. Anderson, J.D.: Fluid of Aerodynamics. McGraw-Hill (2016)
4. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: a comprehensive mechanized proof of a C program. *J. Autom. Reason.* **50**(4), 423–456 (2013)
5. Deeks, A.J., Cheng, L.: Potential flow around obstacles using the scaled boundary finite-element method. *Int. J. Numer. Meth. Fluids* **41**(7), 721–741 (2003)
6. Deniz, E.: Formalization of the Potential Flows and the Laplace Equation, HOL Light Script (2024). <https://hvg.ece.concordia.ca/code/hol-light/pde/le/potential.flows.ml>
7. Deniz, E., Rashid, A., Hasan, O., Tahar, S.: On the formalization of the heat conduction problem in HOL. In: Intelligent Computer Mathematics, LNCS, vol. 13467, pp. 21–37. Springer (2022)

8. Deniz, E., Rashid, A., Hasan, O., Tahar, S.: Formalization of the telegrapher's equations using higher-order-logic theorem proving. *J. Appl. Logics* **11**(2), 197–236 (2024)
9. Dragos, L.: *Mathematical Methods in Aerodynamics*. Kluwer Boston Incorporated (2004)
10. Evans, L.C.: *Partial Differential Equations*. American Mathematical Society (2022)
11. Gordon, M.J.: HOL: A proof generating system for higher-order logic. In: *VLSI Specification, Verification and Synthesis*, pp. 73–128. Springer (1988)
12. Harrison, J.: *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press (2009)
13. Houghton, E.L., Carpenter, P.W.: *Aerodynamics for Engineering Students*. Elsevier (2003)
14. Kaushik, M.: *Theoretical and Experimental Aerodynamics*. Springer, Singapore (2019). https://doi.org/10.1007/978-981-13-1678-4_14
15. Millikan, C.B.: *Aerodynamics of the Airplane*. Courier Dover Publications (2018)
16. Spurk, J., Aksel, N.: *Fluid Mechanics*. Springer Science & Business Media (2007)
17. Strauss, W.A.: *Partial Differential Equations: An Introduction*. Wiley (2007)
18. Temam, R.: *Navier–Stokes Equations: Theory and Numerical Analysis*, vol. 343. American Mathematical Society (2024)
19. Tritton, D.J.: *Physical Fluid Dynamics*. Springer (2012)



On-the-Fly Proof-Based Verification of Reachability in Autonomous Vehicle Controllers Relying on Goal-Aware RSS

Peter Rivière¹(✉), Tsutomu Kobayashi²(✉), Neeraj Kumar Singh¹(✉), Fuyuki Ishikawa³(✉), Yamine Aït Ameer¹(✉), and Guillaume Dupont¹(✉)

¹ INPT-ENSEEIH/IRIT, University of Toulouse, Toulouse, France
{peter.riviere, nsingh, yamine, guillaume.dupont}@enseeiht.fr

² Japan Aerospace Exploration Agency, Tsukuba, Japan
kobayashi.tsutomu@jaxa.jp

³ National Institute of Informatics, Tokyo, Japan
f-ishikawa@nii.ac.jp

Abstract. Autonomous vehicles (AVs) are expected to satisfy not only safety, but they also shall achieve specific goals, e.g., stopping at a particular location on a shoulder lane of a highway for an emergency evacuation while avoiding collisions. The Goal-Aware Responsibility-Sensitive Safety (GA-RSS) framework was proposed to derive control strategies guaranteed to satisfy safety and goal achievement. This framework extends RSS rules, originally designed for safety in basic traffic situations, with a program logic allowing to reason on goal achievements in complex situations. In [11], the Event-B proof-based formal method was used to design a correct-by-construction model of the whole AV controller with a safety architecture and control strategies derived with the GA-RSS framework. This work is extended to handle liveness properties, which are extensively used to model complex goals and achieving them employing the EB4EB reflexive meta-modelling framework. As a result, relying on the EB4EB meta-model, an on-the-fly verification of temporal properties such as deadlock-freeness and goal reachability has been formalised and performed for advanced reasoning. Furthermore, the case study demonstrates additional strengths of the EB4EB meta-modelling approach, such as improvement of modelling and proof understandability and reusability.

Keywords: Autonomous vehicles · Goal-Aware RSS · Reachability · Proof-based verification · EB4EB framework · Safety architecture · Event-B

1 Introduction

Context. For a long time, industries have been relying on formal methods techniques to design safety critical systems [8]. For the particular case of safety assur-

The authors thank the ANR-19-CE25-0010 *EBRP: EventB-Rodin-Plus project*.

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
K. Ogata et al. (Eds.): ICFEM 2024, LNCS 15394, pp. 314–331, 2024.

https://doi.org/10.1007/978-981-96-0617-7_18

ance of autonomous vehicles (AVs), formal methods and particularly theorem proving based methods are effective, since proofs serve as *strong and explainable evidence*, and assist in the core development process to identify faults in early stages of the system development. At least, using theorem proving to guarantee the AV's safety under certain conditions would help to blame the environment to some extent for a good reason.

The Responsibility-Sensitive Safety (RSS) [20] framework is a promising approach to safety for AV systems, that can be used with theorem proving methods to establish complex functional safety properties of AV controllers. It provides formalised rules AVs should follow for safety (i.e., no collision) in basic traffic situations. However, realistic AVs are expected to satisfy not only safety but also *goal achievement*, e.g., reaching an exit on a highway at a low speed while avoiding collisions. As an extension of RSS, the Goal-Aware RSS (GA-RSS) framework [9] is proposed to guarantee both goal achievement and safety in complicated driving scenarios.

In [11], correct-by-construction formal models of AV controllers that follow the rules derived in GA-RSS are derived. By using a safety architecture, the controller is guaranteed to be safe even if the controllers embed black-box components (e.g., components based on machine learning). In this work, the Event-B method [1], which supports an *on-the-fly proof* while developing models at multiple abstraction levels, is used. This approach provides a set of proof obligation rules for safety properties of a model and for the simulation relation between two models (refinement). Furthermore, it enables the step-wise construction of complex models – from abstract models to concrete models – while checking their correctness by discharging proof obligations generated from the rules every time a model is constructed.

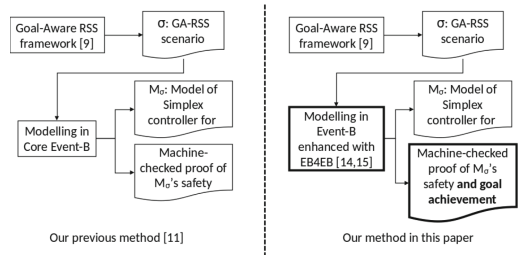


Fig. 1. The revised method in this paper is an extension of previous work [11]

Objective. Starting from the work of [11] which formalised a set of properties required by GA-RSS and demonstrated their safety implications, the objectives of this paper are twofold. The first objective is to extend [11] to have machine-checked proof of goal achievement formalised as a *liveness* case (new contributions are highlighted in dark rectangular boxes in Fig. 1). Here, the difficulty comes from the fact that *Event-B does not support explicit formalisation of liveness properties* nor theorem proving for such domain-specific properties. The second objective consists in performing liveness properties verification in the *same framework provided by Event-B*, avoiding model transformation, and the certification of the transformation to other formal verification techniques like model checking.

Relying on the work in [15] and the developed models, the proposal is to use a meta-modelling approach grounded in the EB4EB framework [14] to extend Event-B with liveness while also considering domain specific scenarios such as GA-RSS goals, the whole developments being conducted in a single framework.

Organisation of this Paper. The remainder of this paper is organised as follows: Sect. 2 presents related work, and the Event-B modelling framework is described in Sect. 3. The GA-RSS framework is introduced in Sect. 4 while Sect. 5 describes the EB4EB meta-modelling framework. Section 6 describes the initial solution with classical Event-B and then the instantiation of the EB4EB framework is presented for verification purpose. Section 7 describes the extension of EB4EB with new properties (i.e., liveness) to complement verification. Finally, Sects. 8 and 9 provide assessment and conclusion with identified perspectives, respectively.

2 Related Work

Various formal approaches have been proposed for the safety of AVs. In [10], RSS rules are encoded as STL formulas for monitoring AVs. Such lightweight formal methods are helpful, but more thorough verification using formal models is also studied for strong guarantees. Roohi et al. [17] constructed a formal model based on RSS and argued that existing fully automatic tools (e.g., reachability analysis tools or model checkers) are unsuitable for verifying such models. Several studies take theorem proving approaches with RSS. Selvaraj et al. [18] verified the safety of AVs with KeYmaera X. Crisafulli et al. [4] considered interactions between actors and used HOL-CSP to formalize and verify the safety of several traffic scenarios. Rizaldi et al. [16] used Isabelle/HOL to prove safety and goal reachability in a simple white-box AV model in simple driving scenarios. Eberhart et al. [6] defined an extension of the program logic of GA-RSS to verify properties of the whole safety architecture that include black-box components.

This paper aims to derive formal models of AV controllers that include black-box components used in complex driving scenarios. The prime aim is to derive correct-by-construction models while performing on-the-fly proof of both safety and goal reachability in a single formal modelling setting grounded in formalised *Event-B extensions*. Currently, only white-box AVs controllers have been modelled using Event-B [5] and hybrid Event-B [2]. In [12], Laibinis et al. proposed a framework to model, within Event-B, goal-oriented multi-agent systems derived from goal-oriented state transition systems defined by the authors. [11] describes the first attempt of addressing black-box AVs controllers using Event-B.

In Event-B, liveness properties can be verified using model checking tools such as ProB [13]. A formal *institution* for Event-B to establish correctness is introduced in [7], facilitating the composition of diverse semantics and model definitions. Recently, a reflexive meta-modelling framework, EB4EB [14, 15], was proposed to perform advanced new proof-based, non-intrusive analyses on Event-B models. It allows to enrich event-B with additional certified proof obligations.

In particular, it assesses reachability, liveness properties, deadlock-freeness, finding weak invariants, and so on.

In this paper, the EB4EB reflexive framework is set up to perform proof-based reachability analysis in AV controllers relying on GA-RSS. This work aims at formalising and verifying complex goal achievements using the EB4EB framework which supports the expression and verification of reachability properties in Event-B models. Furthermore, the goal is to verify the reachability of complex compositional goals *that cannot be reached without first achieving specific subgoals*. This is dependent on the GA-RSS framework for formal backwards reasoning (like in Hoare logic) for identifying such subgoals. Note that this backward reasoning fits well with the backward reasoning style offered by Event-B proof obligations and associated proof system.

3 Event-B: A State Based Formal Method

3.1 Event-B

Table 1. *Global structure of Event-B Contexts, Machines and Theories*

Context	Machine	Theory
CONTEXT Ctx	MACHINE M	THEORY Th
SETS s	SEES Ctx	IMPORT Th1, ...
CONSTANTS c	VARIABLES x	TYPE PARAMETERS E, F, \dots
AXIOMS A	INVARIANTS $I(x)$	DATATYPES
THEOREMS T_{ctx}	THEOREMS $T_{mch}(x)$	Type1 (E, \dots)
END	VARIANT $V(x)$	constructors
	EVENTS	cstr1 ($p_1: T_1, \dots$)
	EVENT evt	OPERATORS
	ANY α	Op1 <nature> ($p_1: T_1, \dots$)
	WHERE $G_i(x, \alpha)$	well-definedness $WD(p_1, \dots)$
	THEN	direct definition D_1
	$x : BAP(\alpha, x, x')$	AXIOMATIC DEFINITIONS
	END	TYPES A_1, \dots
	...	OPERATORS
	END	AOp2 <nature> ($p_1: T_1, \dots$): T_r
		well-definedness $WD(p_1, \dots)$
		AXIOMS A_1, \dots
		THEOREMS T_1, \dots
		PROOF RULES R_1, \dots
		END

(a)

(b)

(c)

Event-B [1] is a state-based formal modelling method that supports a *correct-by-construction* approach. In this method, the system's behaviour is represented by a collection of events that describe state transitions. The language is based on first-order logic (FOL) and set theory.

Modelling. *Contexts* (Table 1.a) and *Machines* (Table 1.b) are two main modelling components, with *Contexts* describing the model’s static part: *carrier sets* s , *constants* c , *axioms* A and *theorems* T_{ctx} , and *Machines* describing the dynamic parts: *variables* x , *invariants* $I(x)$, *variants* $V(x)$, *events* evt and *theorems* $T_{mch}(x)$. Each event, which can be guarded by *guard* G and/or parameterized by *parameters* α , models state variables evolution using a Before-After Predicate (BAP). *Invariants* $I(x)$ and *theorems* $T_{mch}(x)$ are used to encapsulate safety properties, and *variants* $V(x)$ can be generated to ensure machine’s convergence.

Table 2. *Contexts and Machines proof obligations*

(1) Ctx Theorems (ThmCtx)	$A(s, c) \Rightarrow T_{ctx}$ (For contexts)
(2) Mch Theorems (ThmMch)	$A(s, c) \wedge I(x) \Rightarrow T_{mch}(x)$ (For machines)
(3) Initialisation (Init)	$A(s, c) \wedge BAP(x') \Rightarrow I(x')$
(4) Invariant preservation (Inv)	$A(s, c) \wedge I(x) \wedge G(x, \alpha) \wedge BAP(x, \alpha, x') \Rightarrow I(x')$
(5) Event feasibility (Fis)	$A(s, c) \wedge I(x) \wedge G(x, \alpha) \Rightarrow \exists x' \cdot BAP(x, \alpha, x')$
(6) Variant progress (Var)	$A(s, c) \wedge I(x) \wedge G(x, \alpha) \wedge BAP(x, \alpha, x') \Rightarrow V(x') < V(x)$

Proof Obligations. Several proof obligations (POs), shown in Table 2, are associated to Event-B models. These POs relate to theorem consistency ((1) for contexts and (2) for machines), invariant preservation (inductively defined in (3) and (4)), event feasibility (5), and variant decrease (6). All of these POs are generated automatically in the Rodin platform, and must be proven to guarantee the correctness of an Event-B model.

Refinement. One key advantage of Event-B is its *refinement* operation, which allows for the gradual addition of information and details to transform an abstract model into a more concrete one. This process retains a similar observational behaviour (simulation relation) by incorporating refined states and events. Gluing invariants, which describe relationships between states of an abstract model and its concrete model, aid in verifying the correctness of the refinement by ensuring that properties are maintained as it moves from the abstract model to the concrete model.

The Rodin Platform. Rodin is an open source framework for developing and verifying Event-B models. It also supports model checking and animation with ProB, as well as code generation. Rodin also allows the integration of many external provers related to FOL and SMT solvers to aid in the proof process. Moreover, Rodin enables the development of new plug-ins, such as the theory extension.

3.2 Theories Extension

Based on set theory and FOL, the Event-B formalism is mathematically low-level and thus very expressive. However, it lacks features to build up more complex structures. In order to define more complex structures in Event-B, a theory extension [3] (see Table 1.c) is proposed to define additional generic algebraic datatypes together with constructive and axiomatic operators, including well-definedness (WD) conditions, theorems, axioms and new proof rules. The consistency of the resulting theories can be demonstrated by supplying witnesses for axioms and definitions, hence ensuring Event-B extensions. The elements of a theory can be easily accessed in an Event-B model and its proofs after they have been defined. This extension is akin to modularisation mechanisms available in other theorem provers such as Coq, Isabelle/HOL or PVS. Several theories have been defined for real numbers, lists, groups, differential equations, domain-specific theories, etc.

4 The Goal-Aware RSS Framework

4.1 Responsibility-Sensitive Safety (RSS)

Responsibility-Sensitive Safety (RSS) [20] is an approach for guaranteeing AV safety with formal proofs. RSS defines the responsibility of participants in several basic traffic situations for safety (no collisions) so that safety is guaranteed if traffic participants follow the rules. For instance, Fig. 2 shows a situation where the subject vehicle (SV) goes behind a principal other vehicle (POV) on a one-way road.

In this case, SV is responsible for maintaining the distance no less than the minimum safety distance d_{RSS} by braking with a braking rate of more than $a_{brake,min}$ when the distance reaches d_{RSS} (such behaviour is called a *proper response*). d_{RSS} is defined by¹: $d_{RSS}(v_r, v_f) =$

$\max\left(0, \frac{v_r^2}{2a_{brake,min}} - \frac{v_f^2}{2a_{brake,max}}\right)$. Here, v_r and v_f are velocities of the rear vehicle and the front vehicle, respectively, and $a_{brake,min}$ and $a_{brake,max}$ are the normal braking rate and emergency braking rate, respectively.

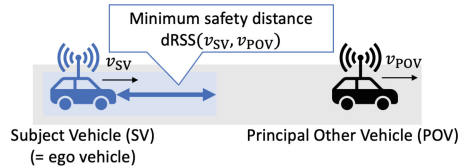


Fig. 2. The one-way traffic scenario [20]

¹ This definition of d_{RSS} is simplified. In RSS [20], the response time (i.e., the time it takes for the vehicle to activate the brake) is also taken into account.

4.2 Goal Achievement in Autonomous Driving

While RSS provides foundations for formal safety assurance in various traffic situations, expectations for AVs include *goal achievement* as well. For instance, Fig. 3 shows a scenario (*pull over* scenario) where the subject vehicle on a highway aims at an emergency stop at a certain place on the shoulder lane without colliding [9]. RSS is not enough for this scenario because:

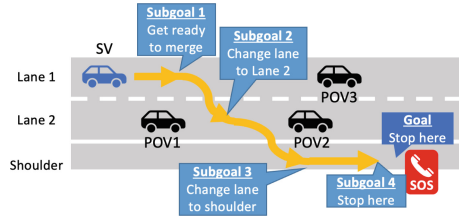


Fig. 3. Pull over scenario [9]

- It should ensure the *goal achievement*, i.e., a certain goal is achieved while maintaining the safety; using only RSS may make the SV trapped in Lane 1.
- There are multiple *subgoals*. To achieve the goal, the SV must achieve certain subgoals while preparing for subsequent subgoals. For instance, the SV should be slowed down once Subgoal 3 is achieved to stop at the goal position.
- Subgoals may have *trade-offs*. For example, the SV should accelerate for Subgoal 1, but accelerating too much prevents achieving Subgoal 4.

For a given situation and goal, an AV controller should be able to select an appropriate sequence of proper responses to achieve this goal.

4.3 Goal-Aware RSS (GA-RSS)

The Goal-Aware RSS (GA-RSS) framework [9] was proposed to identify which sequence of manoeuvres an AV should take in which situation to fulfil the safety and the goal achievement in complicated cases like the pull over scenario.

The core idea of the framework stems from the *similarity between sequential driving manoeuvres and program code*; It can view (sub)goals as postconditions and proper responses as program commands, then derive the precondition of the whole scenario through backward reasoning as in Floyd-Hoare logic.

The framework provides a program logic for such formal discussions named dFHL. It has derivation rules for reasoning asserted hybrid programs (Hoare quadruples) in the form of $\{P\} \alpha \{Q\} : S$, where P and Q are precondition and postcondition, respectively, α is the proper response written as a hybrid program, and S is the safety invariant maintained during the execution of α .

The framework also provides the following tool-assisted workflow for deriving rules for compositional scenarios. The core part of the workflow has four steps for identifying the whole scenario, sub-scenarios, a proper response for each sub-scenario, and a precondition of each sub-scenario.

Step 1: Identifying the Whole Scenario. First, the user identifies the whole scenario of interest. Concretely, (1) the goal condition, (2) the environmental condition, and (3) the safety condition are identified. For instance, the whole pull over scenario is formalised as : $Goal = (l = 3 \wedge x = x_{tgt} \wedge v = 0)$,

$\text{Env} = \bigwedge_{i=1,2,3} (v_{\min} \leq v_i \leq v_{\max} \wedge a_i = 0) \wedge l_1 = 2 \wedge l_2 = 2 \wedge l_3 = 1 \wedge x_2 > x_1$ and $\text{Safe} = \bigwedge_{i=1,2,3} (\text{aheadSL}_i \Rightarrow x_i - x > \text{dRSS}(v_i, v)) \wedge 0 \leq v \leq v_{\max} \wedge -a_{\text{brake},\min} \leq a \leq a_{\max}$.

Here, l and $l_{\{1,2,3\}}$ are lane numbers of SV and three POVs; x and $x_{\{1,2,3\}}$ are their positions; v and $v_{\{1,2,3\}}$ are their velocities; a and $a_{\{1,2,3\}}$ are their accelerations; x_{tgt} is the target position; v_{\min} and v_{\max} are the minimum and the maximum velocity allowed on the highway; $a_{\text{brake},\min}$ is the normal braking rate; a_{\max} is the maximum acceleration of vehicles; and aheadSL_i is whether the i th POV is ahead of the SV in the same lane.

Step 2: Identifying Sub-scenarios. Next, the user decomposes the whole scenario as a sequence of sub-scenarios². For example, from the pull over scenario, a sequence of four sub-scenarios can be identified (Fig. 3). For instance, Sub-scenario 4 is as follows: $\text{Goal}_4 = (x = x_{\text{tgt}} \wedge v = 0)$, $\text{Env}_4 = \text{Env}$ and $\text{Safe}_4 = (l = 3 \wedge 0 \leq v \leq v_{\max} \wedge -a_{\text{brake},\min} \leq a \leq a_{\max})$.

Step 3: Identifying Proper Responses. Then, a proper response is identified for each sub-scenario. For instance, a proper response for sub-scenario 4 (cruise until the distance to the target becomes $\frac{v^2}{2a_{\text{brake},\min}}$, then brake with braking rate $a_{\text{brake},\min}$ until the SV stops) is as follows: $\alpha_4 = (a := 0; \text{dwhile}(\frac{v^2}{2a_{\text{brake},\min}} < x_{\text{tgt}} - x) \{ \dot{x} = v, \dot{v} = a \}; a := -a_{\text{brake},\min}; \text{dwhile}(v > 0) \{ \dot{x} = v, \dot{v} = a \};)$.

Note that α_4 describes a hybrid program detailed in [9], which includes assignment ($:=$), sequence ($;$) and the differential while operation $\text{dwhile}(A) \{ \dot{x} = f \}$, where a dynamic system follows differential equations until the condition A becomes false, and \dot{x} represents the derivative of the function x over time.

Step 4: Identifying Preconditions. Finally, the user identifies the precondition for each sub-scenario from the final sub-scenario to the first through backward reasoning. The precondition should guarantee the achievement of the subgoal and the next precondition after executing the proper response.

For example, a precondition of sub-scenario 4, which satisfies $\{ \phi_4 \} \alpha_4 \{ \text{Goal}_4 \} : (\text{Safe}_4 \wedge \text{Env}_4)$, is as follows: $\phi_4 = (\text{Env} \wedge l = 3 \wedge v > 0 \wedge \frac{v^2}{2a_{\text{brake},\min}} \leq x_{\text{tgt}} - x)$.

The precondition ϕ_4 is used to derive the precondition of sub-scenario 3 (ϕ_3) such that $\{ \phi_3 \} \alpha_3 \{ \phi_4 \wedge \text{Goal}_3 \} : (\text{Safe}_3 \wedge \text{Env}_3)$. This process is repeated until the precondition of the first sub-scenario (ϕ_1), i.e., the precondition of the whole scenario, which can be used to judge if the current situation is an instance of the scenario and SV can take the identified proper responses.

² In general, a *tree* of sub-scenarios is identified by combining multiple decompositions.

4.4 The Simplex Architecture

The Simplex architecture [19] is a software architecture that can be used to ensure the safety of high-performance black-box controllers. It is composed of the *advanced controller* (AC), the *baseline controller* (BC), and the *decision module* (DM) (Fig. 4). AC is typically a black-box controller (e.g., one based on machine learning) designed for various kinds of performance (e.g., comfort and fuel efficiency). BC is a rule-based white-box controller that focuses on safety. DM uses BC if the safety is about to be violated, while it uses AC if the situation is far from a safety violation.

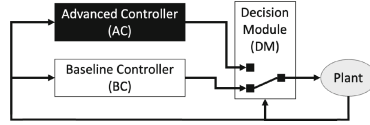


Fig. 4. Simplex architecture [19]

Proper responses derived via the GA-RSS workflow are meant to be used as rules for BC in the Simplex architecture so that the whole controller can provide various kinds of performance while guaranteeing safety and goal achievement.

5 A Formal Meta-model for Event-B: The EB4EB Framework

The EB4EB framework [14] extends Event-B’s formal reasoning process by making available core Event-B elements via meta and reflexive modelling notions. This paradigm enables the development of new reasoning mechanisms for liveness, reachability, deadlock-freeness, etc., which are not supported in core Event-B. This framework takes the form of a set of theories to handle Event-B components using the Event-B theory extension. To model Event-B components, operators and necessary datatypes, together with well-definedness conditions, are defined.

Machine Structure. In Listing 1, a machine is represented by a datatype `Machine` with generic types for events (`Ev`), states (`St`), and parameters (`Pa`). Furthermore, a constructor `Cons_machine` is defined to collect machine components such as `Event`, `State`, `Init`, `Variant`, `BAP`, and so on.

Well-Construction. Although a machine, defined using the machine constructor `Cons_machine`, is syntactically correct, it might be inconsistent. To overcome this issue, an additional set of operators is defined to encode the “well-construction” of a machine and its components. This is a key stage in ensuring the uniformity and correct interaction of the components. For example, the predicate `Event_WellCons` operator models that events are partitioned

```

THEORY EvtBTheo
TYPE PARAMETERS St, Ev, Pa
DATATYPES Machine (St, Ev, Pa)
CONSTRUCTORS
  Cons_machine(
    Event : P(Ev),
    State : P(St),
    Init : Ev, Progress : P(Ev)
    Variant : P(St x Z),
    AP : P(St),
    BAP : P(Ev x ((Pa x St) x St)),
    Grd : P(Ev x (Pa x St)),
    Inv : P(St),
    ... )
    
```

Listing 1. Machine data type

between initialisation and progress events. All well-construction operators are combined in the defined operator `Machine_WellCons`.

Machine POs. Mirroring the Event-B method, the EB4EB framework defines a collection of new operators to encode the proof obligations defined in core Event-B (see Listing 2). All of these operators are predicates that rely on the machine's set-theoretical specification and guarded transition system semantics. For example, for a given machine m the predicate `Mch_INV(m)` holds if and only if the invariants of m hold with regard to m 's behaviour, equivalent to the proof obligation for invariant preservation (`Inv`).

```

Mch_INV_Init <predicate> (m : Machine(St, Ev, Pa))
  direct definition AP(m) ⊆ Inv(m)
Mch_INV_One_Ev <predicate> (m : Machine(St, Ev, Pa), e : Ev)
  well-definedness e ∈ Progress(m)
  direct definition BAP(m)[{e}][{(Pa × Inv(m)) ∩ Grd(m)[{e}]}] ⊆ Inv(m)
Mch_INV <predicate> (m : Machine(St, Ev, Pa))
  direct definition
    Mch_INV_Init(m) ∧ (∀e · e ∈ Progress(m) ⇒ Mch_INV_One_Ev(m, e))
...

```

Listing 2. Well-defined data type operators (behavioural semantics)

Furthermore, all the PO operators are grouped together in a conjunctive form to define a new operator called `check_Machine_Consistency` (see Listing 3) for ensuring the correctness conditions of a machine. This operator features a WD condition enforcing the machine's correct construction, and is typically used as a theorem during theory instantiation. Discharging the proof obligations revolves on stating the `check_Machine_Consistency` predicate operator as a theorem in an Event-B context, instantiating the `Machine` data-type, where the *ThmCtx* PO of Table 2 and the WD PO of the operator are automatically generated.

```

check_Machine_Consistency <predicate>
  > (m : Machine(St, Ev, Pa))
  well-definedness
    Machine_WellCons(m)
  direct definition Mch_INV(m) ∧ ...

```

Listing 3. Operator for Event-B machine consistency

Temporal and Liveness Properties. In [15], the EB4EB framework is extended by another theory for reasoning about temporal and liveness properties in Event-B. The base theory of EB4EB is extended with a set of new operators, WD conditions and POs to cover a range of temporal operators, such as *TLUntil* (\mathcal{U}), *TLPersistence* ($\diamond\Box$) and *TLExistence* ($\Box\diamond$). As they are invoked in a theorem, the proof obligation associated to each operator is generated automatically. It should be noted that the soundness of the given temporal operators is checked using Event-B machines' trace semantics.

Instantiation of EB4EB. The meta theory of EB4EB is used to define specific Event-B machines (instantiation) using the `Cons_machine` constructor. An Event-B context can be used to instantiate the generic types parameters `St`, `Ev` and `Pa`, as well as machine components. Note that new theorems are introduced for both the conventional Event-B POs and the new POs related to temporal properties. These theorems generate new POs that must be discharged to ensure the correctness of the instantiated Event-B model.

6 On-the-Fly Proof for GA-RSS AV Controller Models

This section presents the first contribution showing that all the features of the Goal-RSS Event-B models developed in [11] can be represented in a single Event-B context without loss of expressiveness.

6.1 The Target Model Constructed with Core Event-B

The GA-RSS framework ensures controllers' safety and goal achievement based on rules derived from the framework. However, for controllers constructed with a safety architecture such as Simplex, it is important to ensure the safety and goal achievement of the whole controller. Specifically, the DM should switch between AC and BC to ensure the overall controller's safety and goal achievement.

In previous work [11], Event-B is used to derive a correct-by-construction model of the whole controller that uses the Simplex architecture and GA-RSS rules for each sub-scenario of the pull over scenario. For example, the machine of the controller for sub-scenario 4 (Sect. 4.3) is in Listing 4. The following variables are declared: the position and the velocity of the SV (xSV and vSV), the active controller module ($ctrl$, whose value is ac or bc), the velocity when BC got activated ($vSVBCInit$), and the remaining time for cruising and braking ($tCruise_BC$ and $tBrake_BC$).

For each case, the controller's behaviour is written as an event. For example, event `BC_CruisettoAC` is the behaviour for the case where the current active controller is BC (guard `BC_operating`), the vehicle should still cruise (guard `still_cruise`), and the situation is far from a violation of the precondition ϕ_4 (i.e., ϕ_4 will hold even if the vehicle will keep accelerating with the maximum acceleration rate for two cycles) (guard `in_safety_env_next`). Then, the SV's velocity after a cycle (parameter $vSVp$) will be the same as before (guard `cruise_v` and action `vSV_update`), the SV's position after a cycle (parameter $xSVp$) will be changed due to the constant speed movement (guard `cruise_x` and action `xSV_update`), AC will become active (action `switch_to_AC`), and the remaining time for cruising will be decreased (action `tCruise_pass`).

```

MACHINE subscenario4_2 ...
VARIABLES xSV vSV ctrl vSVBCInit
            tCruise_BC tBrake_BC
INVARIANTS
  precondition: precondition(xSV  $\mapsto$  vSV) = TRUE
  safety_vSV: 0  $\leq$  vSV  $\leq$  vMax
  ...
EVENTS
  Init ...
  BC_CruisettoAC
  ANY xSVp, vSVp
  WHERE
    BC_operating: ctrl = bc
    in_safety_env_next:
      precondition(xSV  $\mp$  (2 * (vSV + aMax))
         $\mapsto$  vSV + (2 * aMax)) = TRUE
    still_cruise: 1  $\leq$  tCruise_BC
    will_brake: tBrake_BC  $\neq$  0
    have_been_cruising:
      vSV = vSVBCInit
    cruise_v: vSVp = vSVBCInit
    cruise_x: xSVp = xSV + vSVBCInit
  THEN
    xSV_update: xSV := xSVp
    vSV_update: vSV := vSVp
    switch_to_AC: ctrl := ac
    tCruise_pass:
      tCruise_BC := tCruise_BC - 1
  END
  BC_Brake_runtoAC ...
  ...

```

Listing 4. Sub-scenario 4 as an Event-B machine [11]

Formal verification is performed to ensure the preservation of invariant predicates. The main invariant is the precondition of sub-scenario 4 (ϕ_4) (invariant **precond**), i.e., the whole controller (including black-box AC) satisfies ϕ_4 due to DM's switching. Although the preservation of ϕ_4 and GA-RSS framework guarantees the safety of sub-scenario 4 (**Safe₄**) (invariant **safety_vSV³**), this is also proven to preserve **Safe₄** in the Event-B environment, with machine-checked proof.

While the machine-checked proof of the safety properties was completed, the proof for goal achievement was not completed as core Event-B does not support proof of liveness properties. The rest of the paper describes how this problem is tackled relying on the EB4EB extension capabilities.

6.2 Formal Verification Using EB4EB Framework

This section covers the advanced-level formal reasoning on AVs for dealing with the challenges raised in Sect. 6.1. The developed case study requires support for verifying temporal properties, such as liveness and reachability. The main contribution of this paper is to perform formal verification of such properties using the EB4EB framework. The usage of the EB4EB framework and its temporal theory extension enables the generation of additional proof obligations, particularly those related to liveness and its associated missing key properties.

Note that these key properties have emerged as a result of the integration of several complex fields such as the Simplex architecture, cyber-physical systems and GA-RSS. The EB4EB framework is used to formally verify these properties by encoding them as additional POs. Checking such POs inherits from EB4EB the benefits of non-intrusive analyses of the core model, explicit modelling of new properties, and reusability.

```

CONTEXT subscn4_2_deep EXTENDS ctx_subscn4, ACBC
SETS EvInst CONSTANTS StInst, PaInst, init, BC_CruisetoAC, ...
AXIOMS
  axm1: partition(EvInst, {init}, {BC_CruisetoAC}, ...)
  axm2-3: StInst = ℝ × ℝ × {ac, bc} × ℝ × ℝ × ℝ ∧ PaInst = ℝ × ℝ
  axm4: subscn4_2 ∈ Machine(StInst, EvInst, PaInst)
  axm5: Grd(subscn4_2) = {ev ↦ ((xSVp ↦ vSVp) ↦
    (xSV ↦ vSV ↦ ctrl ↦ vSVBCInit ↦ tCruise_BC ↦ tBrake_BC)) |
    (ev = BC_CruisetoAC ∧ ctrl = bc ∧ 1 ≤ tCruise_BC ∧ tBrake_BC ≠ 0 ∧
    precondition(xSV + 2 * (vSV + aMax) ↦ vSV + 2 * aMax) = TRUE ∧
    vSV = vSVBCInit ∧ vSVp = vSVBCInit ∧ xSVp = xSV + vSVBCInit) ∨ ...}
  axm6: BAP(subscn4_2) = {ev ↦ (((xSVp ↦ vSVp) ↦ (... ↦ tCruise_BC) ↦
    (xSV' ↦ vSV' ↦ ctrl' ↦ ... ↦ tCruise_BC')) | vSV' = vSVp ∧ xSV' = xSVp ∧
    (ev = BC_CruisetoAC ∧ ctrl' = ac ∧ tCruise_BC' = tCruise_BC - 1) ∨ ...)}
  ...
  thm1: check_Machine_Consistency(subscn4_2)
END

```

Listing 5. Sub-scenario 4 represented as a context

To perform such a formal reasoning, the development of the AV controllers using the so-called *deep modelling instantiation technique* is described. All constituents of the AV controller models are explicitly formulated in terms of the

³ In the model, clauses in **Safe₄** related to l and a are abstracted away.

EvtBTheo theory constructs. The Event-B models of AV controllers such as Listing 4 are represented in an Event-B context (Listing 5), and proof obligations are described either as theorems or as WD conditions.

In this context, the generic parameters, Ev , St , Pa are instantiated with Ev_{Inst} , St_{Inst} , and Pa_{Inst} , respectively. An enumerated set Ev_{Inst} lists all the autonomous vehicle events in $axm1$, a constant St_{Inst} represents autonomous vehicle states defined in axiom $axm2$ as the Cartesian product of the variable types, and another constant Pa_{Inst} represents autonomous vehicle events parameters, in axiom $axm3$. In $axm4$, the autonomous vehicle machine $subscn4_2$ is defined with type $Machine(Ev_{Inst}, St_{Inst}, Pa_{Inst})$. Additional axioms enable the instantiation of various machine components using set comprehensions. For example, $Grd(subscn4_2)$ and $BAP(subscn4_2)$ (defined in $axm5$ and $axm6$) present respectively a set of guards and a collection of BAP predicates for the autonomous vehicle machine $subscn4_2$.

Finally, the theorem $thm1$ is added, consisting of the $check_Machine_Consistency$ predicate, in order to check the WD conditions and POs generation of the autonomous vehicle machine. Note that the RSS goals and subgoals are now encoded in the theorem $thm1$. Indeed, when proving this theorem, these subgoals are highlighted in the backward proof supported by the Rodin platform. They are propagated in the proof tree as post-conditions of the events.

7 On-the-Fly Reachability Analysis

This section discusses advanced level formal reasoning using the EB4EB framework and its temporal extensions. Due to space limitation, only two important properties are presented below: *deadlock-freeness* and *goal reachability*.

7.1 Deadlock-Freeness of AV Controller

The controller's deadlock-freeness is verified as follows:

```

THEORY ControllerDeadlock IMPORT EvtBTheo
TYPE PARAMETERS  $St, Ev, Pa$ 
OPERATORS
DeadLockFreeController <predicate> ( $m : Machine(Ev, St, Pa)$ ,
   $controlEvs : \mathbb{P}(EVENT), Goal : \mathbb{P}(St)$ )
well-definedness condition  $Machine\_WellCons(m) \wedge controlEvs \subseteq Progress(m)$ 
direct definition  $Inv(m) \cap (St \setminus Goal) \subseteq ran(Grd(m)[controlEvs])$ 

```

Listing 6. Theory of deadlock-freeness of the controller

Formalisation. In Listing 6, a new theory **ControllerDeadlock** is introduced, allowing deadlock-freeness properties to be expressed in the context of AV controllers. In this theory, a predicate operator **DeadLockFreeController** is defined with three arguments: m , $controlEvs$, and $Goal$, where m is the AV controller machine, $controlEvs$ is a set of controller events, and $Goal$ is a set of states that satisfy the goal. Furthermore, a WD condition is supplied to ensure that the defined operator is being used correctly. The required WD condition checks

that the machine is well constructed ($Machine_WellCons(m)$) and that the controller events are progress events ($controlEvs \subseteq Progress(m)$). The definition $Inv(m) \cap (St \setminus Goal) \subseteq ran(Grd(m)[controlEvs])$ ensures that at least one controller event of the $controlEvs$ event set is enabled when the system does not satisfy the $Goal$ states while the invariants hold.

Application to the Case Study. The deadlock-freeness operator defined above is used for analysing the pull over case study: a context `subscenario4_2_deep_analyse` extending `subscenario4_2_deep` of is defined, consisting of the theorem:

`DeadlockFreeController(subscn4_2, Progress(subscn4_2), {xSV ↦ vSV ... | vSV = 0})`.

This theorem uses the predicate operator `DeadlockFreeController` previously formalised, in which m is the `subscn4_2` machine, $controlEvs$ is the set of progress events $Progress(subscn4_2)$, and $Goal$ corresponds to stopping the SV at $vSV = 0$. Whenever this theorem is used, a PO is automatically generated that must be proved for ensuring deadlock-freeness of the AV controller.

THEORY <code>TheoGoalIsReached</code>
IMPORT <code>EvtBLiveness</code>
TYPE PARAMETERS <code>St, Ev, Pa</code>
OPERATORS
GoalIsReached < <i>predicate</i> >
($m : Machine(Ev, St, Pa)$,
$Goal : \mathbb{P}(St), v : \mathbb{P}(St \times \mathbb{Z})$)
well-definedness condition
$Machine_WellCons(m) \wedge v \in St \rightarrow \mathbb{Z}$
direct definition
$TLPersistence(m, Goal, v)$

Listing 7. *Goal reachability theory*

7.2 Goal Reachability in AV Controller

The primary intent of this property is to ensure that GA-RSS goals are eventually achieved. With this aim, the formalisation of liveness described in EB4EB [15] is extended to integrate the aspects of GA-RSS as follows:

Formalisation. In Listing 7, a new theory `TheoGoalIsReached` is defined that imports the previously developed liveness theory `EvtBLiveness` to access the operators that define liveness POs. In this theory, a predicate operator `GoalIsReached` is defined with three arguments: m , $Goal$ and v , where m is the autonomous vehicle machine, $Goal$ is a set of states satisfying GA-RSS, and v is a variant. Furthermore, a WD condition is supplied to ensure that the defined operator is used correctly. The required WD condition is to check that the machine is well constructed ($Machine_WellCons(m)$) and the variant v is correctly defined as a total function from STATE to \mathbb{Z} ($v \in St \rightarrow \mathbb{Z}$). The direct definition of the operator relies on the `TLPersistence` ($\diamond\Box$) operator, entailing the property that the goal must eventually hold forever.

Application to the Case Study. A context, `subscenario4_2_deep_analyse`, extending the context `subscenario4_deep` is defined with a theorem:

`GoalIsReached(subscn4_2, {xSV ↦ vSV ↦ ... | vSV = 0}, variant)`

This theorem uses the predicate operator `GoalIsReached`, previously formalised, where machine m is `subscn4_2`, $Goal$ is a set of states satisfying GARSS, i.e. to stop the SV at $vSV = 0$, and $variant$ is the variant. This variant is defined as $variant = \lceil \frac{xTgt - xSV}{vMin} \rceil$ to be the number of cycles to reach the target at the minimum speed. Note that the original development is not precise enough for proving this property. Thus, additional axioms are added, such as the fact that minimum brake power can bring the vehicle to a stop when it is at the minimum speed in one cycle ($vMin < \frac{aBrakeMin}{2}$), which is required to discharge the reachability theorem.

8 Assessment

Proof Factorisation and Simplification. The EB4EB framework allows meta-level manipulation of several Event-B models simultaneously through the deep embedding instantiation mechanism. However, the accompanying proofs for different POs are performed at the classical Event-B machine level. Lifting proofs at the meta level with an advanced factoring method can simplify such proving processes as well as assist in automating proof tree isomorphisms. First, factoring methods allow for proof processes to be performed at a higher level, as represented by an isomorphic proof tree, and then classical proof steps can be applied at a lower level of the proof tree associated to specific events.

For example, in the case study, the goal reachability PO is defined, where the variant must decrease $\lceil \frac{xTgt - xSV'}{vMin} \rceil < \lceil \frac{xTgt - xSV}{vMin} \rceil$. Steps to decrease this inequality to $vMin < xSV' - xSV$ are taken for all events. The granularity can be modified, and the factorisation can be defined for all events or for a subset of events with do-cases. The factoring approach allows the factorisation of ~ 500 nodes in the proof tree for nine events in the case study. Note that the total number of nodes in the *reachability goal* and *deadlock-freeness* POs after factorisation is only 4289 and 4258, respectively, but the original POs of both have more than 8000 nodes. Only 10% of the nodes in the proof correspond to interactive proofs, while the remaining 80% correspond to typing proofs based on Real theory, that can be mechanised easily. The last 10% correspond to automatic nodes derived from Rodin's tactic, also proved automatically.

Model Enhancement. The EB4EB framework can be used for advanced level formal reasoning by adding new safety properties. These new properties would not be planned during the initial development. Thus, some elements of the models may be weak or incomplete during the proof process of new properties, resulting in infeasible proofs. By adding additional invariants or theorems, tightening the guard of events, or adding new hypotheses to the specification, the original models can be improved to make them complete in order to discharge the generated POs. The initial model of the case study is robust enough to prove deadlock-freeness, however the proof of goal reachability reveals the presence of the Zeno problem on the models. To solve this problem, a new requirement was introduced: the actual speed must be greater than the minimum speed of the

vehicle. The guards of the advanced controller events as well as the invariants are modified to match the new requirements for discharging the domain POs

Generating Domain Specific POs. In the selected case study, several new POs are introduced, such as *deadlock-freeness* and *goal reachability*. These POs are not *explicitly* supported in the core Event-B. Their formalisation in Event-B requires the expertise of the model designers and there are no reuse possibilities as they need to be expressed each time a model is developed (manipulating guards, expressing variants and convergent events, etc.). In our approach, these new POs are encoded using the EB4EB framework. Note that the EB4EB framework may encode classical Event-B POs, but it can also be used to extend additional POs linked to temporal concepts as well as domain-specific notions. For example, the position and velocity cannot be negative, thus a new PO can be encoded to generate this domain-specific property. Such POs are useful for encapsulating standards during requirement modelling, and the EB4EB framework also aids in the proving process based on domain-specific properties.

The interest of the approach is the capability to instantiate previously domain specific POs voiding designers to define the POs themselves. We believe that this approach promotes the dissemination of the use of formal methods by engineers.

9 Conclusion

This paper proposed a formal technique that allows a designer to verify domain-specific temporal properties, deadlock-freeness, and goal reachability on AV controllers based on GA-RSS. The extended reasoning mechanism of the reflexive EB4EB framework is used to perform on-the-fly proof-based verification by encoding the system's domain specific requirements. The obtained framework composes different formalised concepts like temporal logic or AVs' domain gathered in a unified setting. Other domain-specific concepts may be included as well, provided that they are formalised as theories to be imported.

At this level, one may ask why the expression and verification of the properties studied in this paper are not achieved with classical Event-B machines. There are two reasons for this. First, while Event-B is better suited for developing system models and ensuring inductive invariant preservation, it does not preserve liveness properties in refinement chains in general. Preserving such properties requires the definition of decreasing variants and guaranteeing their convergence at each refinement level. The same applies for deadlock. There is no automatic proof obligation generation for such properties.

The whole formalisation and verification of sub-scenario 4.1 is illustrated in Event-B using the EB4EB framework and the technique is demonstrated by performing advanced-level proof of goal reachability in different pull over scenarios relying on GA-RSS. This proof was not feasible in previous development due to the lack of support for liveness properties in core Event-B. In this work, all machine-checked proofs are performed in the Event-B environment and the model is instantiated for advanced level liveness properties to get full confidence

in the developed model as well as to use proof artefacts as strong and explainable evidence to meet certification requirements. Due to space limitation, only deadlock-freeness and goal reachability properties are demonstrated in the paper, however all the developments presented in this paper are completely formalized, and all proof obligations have been discharged. These developed models are available on <https://www.irit.fr/~Peter.Riviere/models/>.

There are two imminent challenges identified. The first one is defining concise and precise domain-specific engineering theories to cover every aspect of the GARSS framework in order to define inbuilt operators for designing AV controllers as well as to define new POs to meet systems goals. Additionally, this approach can be exploited for certification purposes to meet industry safety standards for AVs. Here, we target the definition of generic proof obligations formalising AVs controllers standards.

References

1. Abrial, J.R.: *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge (2010)
2. Banach, R., Butler, M.J., Qin, S., Verma, N., Zhu, H.: Core hybrid event-b I: single hybrid event-b machines. *Sci. Comput. Program.* **105**, 92–123 (2015)
3. Butler, M.J., Maamria, I.: Practical theory extension in Event-B. In: *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, pp. 67–81 (2013)
4. Crisafulli, P., Taha, S., Wolff, B.: Modeling and analysing cyber-physical systems in HOL-CSP. *Robot. Auton. Syst.* **170**, 104549 (2023)
5. Dupont, G., Aït Ameer, Y., Singh, N.K., Pantel, M.: Event-b hybridation: a proof and refinement-based framework for modelling hybrid systems. *ACM Trans. Embed. Comput. Syst.* **20**(4), 35:1–35:37 (2021)
6. Eberhart, C., Dubut, J., Haydon, J., Hasuo, I.: Formal verification of safety architectures for automated driving. In: *2023 IEEE IV*, pp. 1–8 (2023)
7. Farrell, M., Monahan, R., Power, J.F.: Building specifications in the Event-B institution. *Logical Methods Comput. Sci.* **18**(4) (2022)
8. Fitzgerald, J.S., Bicarregui, J., Larsen, P.G., Woodcock, J.: Industrial deployment of formal methods: trends and challenges. In: *Industrial Deployment of System Engineering Methods*, pp. 123–143. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33170-1_10
9. Hasuo, I., et al.: Goal-aware RSS for complex scenarios via program logic. In: *IEEE T-IV*, pp. 1–33 (2022)
10. Hekmatnejad, M., et al.: Encoding and monitoring responsibility sensitive safety rules for automated vehicles in signal temporal logic. In: *17th ACM-IEEE MEM-OCODE*. ACM (2019)
11. Kobayashi, T., Bondu, M., Ishikawa, F.: Formal modelling of safety architecture for responsibility-aware autonomous vehicle via Event-B refinement. In: *FM. LNCS*, vol. 14000, pp. 533–549. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-27481-7_30
12. Laibinis, L., Pereverzeva, I., Troubitsyna, E.: Formal reasoning about resilient goal-oriented multi-agent systems. *Sci. Comput. Program.* **148**, 66–87 (2017)

13. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. Springer Int. J. STTT **10**(2), 185–203 (2008)
14. Riviere, P., Singh, N.K., Ait Ameer, Y.: Reflexive event-B: semantics and correctness the EB4EB framework. IEEE Trans. Reliabil. 1–16 (2022)
15. Riviere, P., Singh, N.K., Ait Ameer, Y., Dupont, G.: Formalising liveness properties in Event-B. In: NFM. LNCS, pp. 312–331. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-33170-1_19
16. Rizaldi, A., Immler, F., Schürmann, B., Althoff, M.: A formally verified motion planner for autonomous vehicles. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 75–90. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_5
17. Roohi, N., Kaur, R., Weimer, J., Sokolsky, O., Lee, I.: Self-driving vehicle verification towards a benchmark. CoRR [arxiv:1806.08810](https://arxiv.org/abs/1806.08810) (2018)
18. Selvaraj, Y., Ahrendt, W., Fabian, M.: Formal development of safe automated driving using differential dynamic logic. IEEE T-IV **8**(1), 988–1000 (2023)
19. Seto, D., Krogh, B., Sha, L., Chutinan, A.: The simplex architecture for safe online control system upgrades. In: 1998 ACC, vol. 6, pp. 3504–3508. IEEE (1998)
20. Shalev-Shwartz, S., Shammah, S., Shashua, A.: On a formal model of safe and scalable self-driving cars. CoRR [arxiv:1708.06374](https://arxiv.org/abs/1708.06374) (2017)



Efficient SMT-Based Model Checking for HyperTWTL

Ernest Bonnah¹, Luan Viet Nguyen², and Khaza Anuarul Hoque³(✉)

¹ Department of Computer Engineering, Baylor University, Waco, TX, USA
ernest_bonnah@baylor.edu

² Department of Computer Science, University of Dayton, Dayton, OH, USA
lnguyen1@udayton.edu

³ Department of Computer Science, University of Missouri, Columbia, MO, USA
hoquek@missouri.edu

Abstract. Hyperproperties extend trace properties to express properties of sets of traces, and thus, they are increasingly popular in specifying various security and performance-related properties in domains such as autonomous, cyber-physical, and robotic systems. Specifically, Hyperproperties for time window temporal logic (HyperTWTL) are known for their compactness in specifying robotic systems' safety and security requirements. However, the existing model checking approach for HyperTWTL verification relies on automata-based model checking, which is computationally expensive and suffers from the state-space explosion problem. This paper introduces a bounded model checking approach for verifying HyperTWTL specifications using SMT solvers. Specifically, our proposed verification method reduces the HyperTWTL model checking problem to a first-order logic satisfiability problem and then uses state-of-the-art SMT solvers, i.e., Z3 and CVC4, for verification. The feasibility of the proposed HyperTWTL verification methods is demonstrated through a Technical Surveillance Squadron (TESS), a Robotic Industrial Inspection case study, and also a scalability analysis. Our results show that the proposed method can offer up to 19× speed up and 2× memory efficiency compared to the traditional automata-based model checking approach. We also show that the proposed HyperTWTL verification technique can verify large systems, whereas the traditional HyperTWTL verification method suffers from state-space explosion problem.

Keywords: Hyperproperties · Bounded Model Checking · Time Window Temporal Logic and Robotics · SMT solver

1 Introduction

Hyperproperties [13] extend the notion of trace properties [1] from a set of traces to a set of sets of traces. This allows specifying a wide range of properties related to information-flow security [20, 32], consistency models in concurrent computing [7, 18], robustness models in cyber-physical systems [6, 19], and also service

level agreements (SLA) [13]. Several types of hyperproperties and their model checking algorithms have been proposed in the recent past, including HyperLTL [12, 14, 17, 23], HyperSTL [25], HyperMTL [8, 21], and HyperTWTL [9]. These formalisms has been successfully used to specify and verify important requirements in different domains including cyber-physical systems, robotics and machine learning. Specifically, for time-bounded and sequential tasks, HyperTWTL offers a rich expressiveness and compactness. For instance, consider a hyperproperty that requires that “for any pair of traces π and π' , A should hold for 5 time steps in trace π within the time bound $[0, 10]$ and B should also hold for 3 time steps in trace π' within the same time bound”. This requirement can be expressed using HyperTWTL formalism as $\varphi = \forall\pi\forall\pi' \cdot [\mathbf{H}^5 A_\pi \wedge \mathbf{H}^3 B_{\pi'}]^{[0,10]}$. The same requirement can be expressed as a HyperSTL formula as $\varphi = \forall\pi\forall\pi' \cdot (\mathbf{F}_{[0,10-5]} \mathbf{G}_{[0,5]} A_\pi) \wedge (\mathbf{F}_{[0,10-3]} \mathbf{G}_{[0,3]} B_{\pi'})$. In HyperMTL this requirement can be expressed as $\varphi = \forall\pi\forall\pi' \cdot \bigvee_{i=0}^{10-5} \mathbf{G}_{[i,i+5]} A_\pi \wedge \bigvee_{i=0}^{10-3} \mathbf{G}_{[i,i+3]} B_{\pi'}$.

HyperTWTL extends the classical Time Window Temporal Logic (TWTL) [30] by allowing explicit and simultaneous quantification over multiple execution traces. The classical approach for verifying HyperTWTL in [9] relies on an automata-based model checking. Traditionally, automata-based model checking is known for its high computation time, memory overhead, and may lead to a state-space explosion. Hence, we propose a more efficient and scalable approach in this paper. Specifically, we propose an SMT-based approach to verify HyperTWTL properties by converting the model checking problem to a first-order logic satisfiability problem. For example, given a HyperTWTL formula $\varphi = \forall\pi_1.\forall\pi_2 \cdot [\mathbf{H}^{15} A_{\pi_1} \wedge \mathbf{H}^{10} B_{\pi_2}]^{[0,20]}$ and a collection of Time Kripke structures (TKS) $\mathcal{M} = \langle M_1, M_2 \rangle$, where M_i is an identical copy of the given Kripke structure mapped to the trace variable π_i , the process to convert both the φ and \mathcal{M} to a first-order logic involves three main steps. First, we compute the unrolling bound $\|\varphi\|$ based on the structure of the formula. Secondly, we encode the path quantifications, initial conditions, the transition relations of each TKS M_i , and the negation of HyperTWTL formula φ as first-order logic formula represented by the encoding $\llbracket M_i \rrbracket_{\|\varphi\|}$ and $\llbracket \neg\varphi \rrbracket_{\|\phi\|}$ respectively. The combination of two encoded formulae is of the form $\llbracket \mathcal{M} \neg\varphi \rrbracket_{\|\varphi\|} = [\exists_1\pi_1] \cdot [\exists_2\pi_2] \cdot \llbracket M_1 \rrbracket_{\|\varphi\|} \wedge \llbracket M_2 \rrbracket_{\|\varphi\|} \wedge \llbracket \neg\phi \rrbracket_{0,\|\varphi\|}$. Lastly, the combined first-order logic formula unrolled to a depth of $\|\varphi\|$ is then solved using an off-the-shelf SMT solver. If the approach returns an affirmative answer, then the SMT solver generates a counterexample. Though the proposed approach is inspired by SMT-based bounded model checking (BMC), as earlier stated, the unrolling bound is calculated using a given HyperTWTL formula. This contrasts with the traditional BMC approach for verifying temporal logic, where an arbitrary unrolling bound is given.

To demonstrate the effectiveness of our approach, we formalize some interesting requirements of two case studies using HyperTWTL. The first case study we consider is a Technical Surveillance Squadron (TESS) [28], known for providing collaborative surveillance of designated regions to detect, identify, and locate potential nuclear explosions. In the second case study, we consider a robotic solu-

tion that automates industrial equipment inspections [2], where robots provide plant operators the information to maximize equipment uptime and improve safety and efficiency. We use two SMT solvers, CVC4 and Z3, both known for their industrial application [4, 29], to compare their performance for verifying the HyperTWTL requirements. Finally, we compare our proposed SMT-based HyperTWTL verification performance with automata-based HyperTWTL verification. We observe that our proposed SMT-based approach offers up to $19\times$ speed up in terms of execution speed and consumes up to $2\times$ less memory when compared to the automata-based HyperTWTL verification approach. We also perform experiments to demonstrate the scalability of our approach and show that we can verify large robotic systems, whereas the automata-based HyperTWTL verification leads to the state-space explosion.

2 Preliminaries

Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{AP}$ be the *alphabet*. We call each member of Σ an *event*. We define a timed trace t as a finite sequence of events from Σ^* , i.e., $t = (\tau_i, e_i), (\tau_{i+1}, e_{i+1}), \dots, (\tau_n, e_n) \in (\mathbb{Z}_{\geq 0} \times \Sigma)^*$ where $\tau_i \tau_{i+1} \dots \tau_n \in \mathbb{Z}_{\geq 0}$ is a sequence of non-negative integers denoting *time-stamps* and the indices $i, n \in \mathbb{Z}_{\geq 0}$ denote *time-points*. We require $\tau_i = 0$, $\tau_i \leq \tau_{i+1}$, and for all i , $0 \leq i \leq n$. For each timed trace t , by $t[i].e$, we mean e_i and by $t[i].\tau$ we mean τ_i . We now define an indexed timed trace as a pair (t, p) where $p \in \mathbb{Z}_{\geq 0}$ is called a *pointer*. Indexed timed traces allow traversing a given trace by moving the pointer. Given an indexed timed trace (t, p) and $m \in \mathbb{Z}_{\geq 0}$, let $(t, p) + m$ denote the resulting trace $(t, p + m)$.

2.1 Kripke Structure

We consider timed systems modeled as timed Kripke structures with the assigned time elapse on the transitions.

Definition 1. A timed Kripke structure (TKS) is a tuple $M = (S, S_{init}, \delta, AP, L)$ where

- S is a finite set of states;
- $S_{init} \subseteq S$ is the set of initial states;
- $\delta \subseteq S \times \mathbb{Z}_{\geq 0} \times S$ is a set of transitions;
- AP is a finite set of atomic propositions; and
- $L : S \rightarrow \Sigma$ is a labelling function on the states of \mathcal{M} .

We require that for each $s \in S$, there exists a successor that can be reached in a finite number of transitions. Hence, all nodes without any outgoing transitions are equipped with self-loops such that $(s, 1, s) \in \delta$. An exemplary TKS is shown in Fig. 1 where $S = \{S_0, S_1, S_2, S_3, S_4, S_5\}$, $S_{init} = \{S_0\}$, $\delta = \{(S_0, 1, S_1), (S_0, 2, S_3), (S_1, 3, S_2), (S_1, 1, S_3), (S_2, 1, S_4), (S_2, 1, S_5), (S_3, 2, S_2), (S_3, 3, S_4), (S_4, 1, S_4), (S_4, 1, S_5), (S_5, 1, S_5)\}$, $L(S_0) = \{\}$, $L(S_1) = \{a, b\}$, $L(S_2) = \{a\}$, $L(S_3) = \{a, b\}$, $L(S_4) = \{a\}$, $L(S_5) = \{a, b\}$ and $AP = \{a, b, c, d\}$. A path over a TKS is a finite sequence of states $S_0 S_1 S_2 \dots S_n \in \Sigma^*$, where $S_0 \in S_{init}$ and $(S_i, d_i, S_{i+1}) \in \delta$, for each $0 \leq i < n$. A trace over TKS is of the form: $t = (\tau_0, e_0)(\tau_1, e_1)(\tau_2, e_2) \dots (\tau_n, e_n)$, such that there exists a path $S_0 S_1 S_2 \dots \in S^*$. Recall an event is of the form (τ_i, e_i) where $\tau_i \in \mathbb{Z}_{\geq 0}$ and $e_i = L(S_i)$.

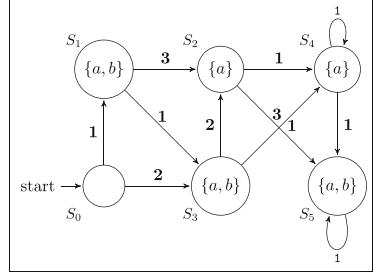


Fig. 1. Timed Kripke structure

2.2 HyperTWTL

HyperTWTL [9] is a hyper-temporal logic to specify hyperproperties for Time Window Temporal Logic (TWTL) [30] by extending TWTL with quantification over multiple and concurrent execution traces. Below we present the syntax of HyperTWTL.

Syntax of HyperTWTL: The syntax of HyperTWTL [9] is inductively defined by the grammar:

$$\begin{aligned} \varphi &:= \exists \pi \cdot \varphi \mid \forall \pi \cdot \varphi \mid \phi \\ \phi &:= \mathbf{H}^d a_\pi \mid \mathbf{H}^d \neg a_\pi \mid \phi_1 \wedge \phi_2 \mid \neg \phi \mid \phi_1 \odot \phi_2 \mid [\phi]^I \mid \mathbf{E}\rho \cdot \psi \mid \mathbf{A}\rho \cdot \psi \\ \psi &:= \mathbf{H}^d a_{\pi, \rho} \mid \mathbf{H}^d \neg a_{\pi, \rho} \mid \psi_1 \wedge \psi_2 \mid \neg \psi \mid \psi_1 \odot \psi_2 \mid [\psi]^{I, J} \end{aligned}$$

where $a \in AP$, π is a trace variable from a set of trace variables \mathcal{V} and ρ is a trajectory variable from the set \mathcal{P} . Thus, given $a_{\pi, \rho}$, the proposition $a \in AP$ holds in trace π and trajectory ρ (explained in Appendix) at a given time point. Trace quantifiers $\exists \pi$, and $\forall \pi$ are interpreted as “there exists some trace π ” and “for all the traces π ”, respectively. Similarly, trajectory quantifiers $\mathbf{E}\rho$ and $\mathbf{A}\rho$ allow reasoning simultaneously about different trajectories. The quantifier $\mathbf{E}\rho$ means there exists at least one trajectory ρ that evaluates the relative passage of time between the traces for which the given inner temporal formula is satisfied. In contrast, $\mathbf{A}\rho$ is interpreted as all trajectories ρ satisfy the inner TWTL formula regardless of the time passage across traces. The operators \mathbf{H}^d , \odot , and $[\]^I$ (as well as $[\]^{I, J}$) represent the hold operator with $d \in \mathbb{Z}_{\geq 0}$, concatenation operator, and within operator respectively, while both I and J are discrete-time constant intervals of form $[\tau, \tau']$, where $\tau, \tau' \in \mathbb{Z}_{\geq 0}$ and $\tau' \geq \tau$, respectively and \wedge and

\neg are the conjunction and negation operators respectively. Trace quantifiers $\exists\pi$ and $\forall\pi$, allow for the simultaneous reasoning about different traces. Given a HyperTWTL formula φ , we denote \mathcal{V}_φ (respectively \mathcal{P}_φ) as the set of trace variables (respectively, trajectory variables) quantified in φ . Thus, we say a given formula φ is closed if for $a_{\pi,\rho}$ in φ , π and ρ are quantified in φ ($\pi \in \mathcal{V}_\varphi$ and $\rho \in \mathcal{P}_\varphi$) and no π and ρ is quantified twice in φ . The disjunction operator (\vee) can be derived from the negation and conjunction operators. Likewise, the implication operator (\rightarrow) can also be derived from the negation and disjunction operators.

Table 1. Synchronous semantics of HyperTWTL

$(\mathbb{T}, II) \models \exists\pi.\varphi$	iff $\exists t \in \mathbb{T} \cdot (\mathbb{T}, II[\pi \rightarrow (t, 0)]) \models \varphi$
$(\mathbb{T}, II) \models \forall\pi.\varphi$	iff $\forall t \in \mathbb{T} \cdot (\mathbb{T}, II[\pi \rightarrow (t, 0)]) \models \varphi$
$(\mathbb{T}, II) \models \mathbf{H}^d a_\pi$	iff $a \notin t[i].e$ for $(t, p) = II(\pi)$, $\forall p \in \{i, \dots, i + d\} \wedge (t[i + n].\tau - t[i].\tau) \geq d$, for some $n > 0$ and $i \geq 0$
$(\mathbb{T}, II) \models \mathbf{H}^d \neg a_\pi$	iff $a \notin t[i].e$ for $(t, p) = II(\pi)$, $\forall p \in \{i, \dots, i + d\} \wedge (t[i + n].\tau - t[n].\tau) \geq d$, for some $n > 0$ and $i \geq 0$
$(\mathbb{T}, II) \models \phi_1 \wedge \phi_2$	iff $((\mathbb{T}, II) \models \phi_1) \wedge ((\mathbb{T}, II) \models \phi_2)$
$(\mathbb{T}, II) \models \neg\phi$	iff $\neg((\mathbb{T}, II) \models \phi)$
$(\mathbb{T}, II) \models \phi_1 \odot \phi_2$	iff $\exists i, j, k$ s.t. $i \leq k \leq j$ and $k = \min k' \mid i \leq k' \leq j, (\mathbb{T}_{[i,k]}, II) \models \phi_1 \wedge ((\mathbb{T}_{[k+1,j]}, II) \models \phi_2)$ for some $i, j \geq 0$
$(\mathbb{T}, II) \models [\phi]^{x,y}$	iff $\exists i, j, k$ and $k \geq i + x$, s.t. $(\mathbb{T}_{[k,i+y]}, II) \models \phi \wedge ((II)^j - (II)^{now}) \geq y$ for some $i, j \geq 0$

Semantics of HyperTWTL: The semantics of HyperTWTL [9] can be divided into *synchronous* and *asynchronous* based on the timestamps in all quantified traces that match at each point in time or proceed at different speeds, respectively. We denote the set of trace variables used in a given HyperTWTL formula φ as \mathcal{V}_φ . We define a collection of copies of TKS as $\mathcal{M} = \langle M_i \rangle_{\pi_i \in \mathcal{V}_\varphi}$, where each M_i is an identical copy of a given TKS used to represent path π_i . We therefore denote a set of traces over \mathcal{M} as \mathbb{T} . Thus, for any given HyperTWTL formula φ , we interpret $\mathbb{T} = \langle T_{\pi_i} \rangle_{\pi_i \in \mathcal{V}_\varphi}$ as the tuple of sets of traces with a set T_{π_i} assigned to $\pi_i \in \mathcal{V}_\varphi$. Thus, for a given collection of TKS \mathcal{M} , we define T_{π_i} as the set of traces over the trace variable π_i coming from M_i . For any given set of sets of traces denoted as $\mathbb{T}_{[i,j]}$, we say the evaluation of all the traces in \mathbb{T} against a formula starts from the time-point $i \geq 0$ up to and including the time-point $j \geq i$. Both semantics of HyperTWTL are presented below.

Synchronous Semantics of HyperTWTL: We define an assignment $II : \mathcal{V} \rightarrow (\mathbb{Z}_{\geq 0} \times \Sigma)^* \times \mathbb{Z}_{\geq 0}$ as a partial function mapping trace variables to time-stamped traces. Let $II(\pi) = (t, p)$ denote the time-stamped event from trace t at position

p currently employed in considering trace π . We then denote the explicit mapping of the trace variable π to a trace $t \in \mathbb{T}$ at position p as $\Pi[\pi \rightarrow (t, p)]$. Thus, by $\Pi(\pi) = (t, p)$, we mean the event from the timed trace t at the position p is currently used in the analysis of trace π . Given the mapping Π , we use $(\Pi) + k$ as the k^{th} successor of Π , i.e., the k^{th} timed event of a mapped trace reached after moving k steps across Π . The hold operator $\mathbf{H}^d a_\pi$ states that the proposition a will be repeated for d time units in trace π . Similarly $\mathbf{H}^d \neg a_\pi$, requires that for d time units the proposition a should not occur in trace π . The trace set \mathbb{T} satisfies both sub-formulae in $\phi = \phi_1 \wedge \phi_2$ while in $\neg\phi$, \mathbb{T} does not satisfy the given formula. A given formula with a concatenation operator in the form $\phi = \phi_1 \odot \phi_2$ specifies that every $t \in \mathbb{T}$ should satisfy ϕ_1 first and then immediately ϕ_2 must also be satisfied with one-time unit difference between the end of execution of ϕ_1 and the start of execution of ϕ_2 . The trace set \mathbb{T} must satisfy ϕ between the time window within the time window $[\tau, \tau']$ given $\phi = [\phi]^{[\tau, \tau']}$. Given Π , we define the the current instant denoted as $(\Pi)^{\text{now}}$ and the j^{th} instant denoted as $(\Pi)^j$ as follows [8]:

$$\begin{aligned}
 (\Pi)^{\text{now}} &= \max_{\pi \in \text{dom}(\Pi)} \{t[p].\tau \mid \text{for } \Pi(\pi) = (t, p)\} \\
 (\Pi)^j &= \min_{\pi \in \text{dom}(\Pi)} \{t[p+j].\tau \mid \text{for } \Pi(\pi) = (t, p)\}
 \end{aligned}$$

We say a collection of traces \mathbb{T} generated over a collection of TKS \mathcal{M} satisfies a synchronous HyperTWTL formula φ if $(\mathbb{T}, \Pi) \models_s \varphi$. We present the synchronous semantics of HyperTWTL in Table 1.

Asynchronous Semantics of HyperTWTL: To define the Asynchronous semantics of HyperTWTL, we adopt the concept of trajectory as used in [23]. For a given HyperTWTL formula, a trajectory $v = v_i v_{i+1} v_{i+2} \dots$ is a sequence of subsets of \mathcal{P}_φ , i.e. $v_i \subset \mathcal{P}_\varphi, \forall i \geq 0$. We call a trajectory a fair trajectory if, for a trace variable $\pi \in \mathcal{P}_\varphi$, there are infinitely many positions i such that $\pi \in v_i$. We denote $\mathcal{R}_\mathcal{P}$ as the set of all fair trajectories for indices from the set of trajectories \mathcal{P} . We now define the trajectory mapping $\Gamma : \mathcal{P}_\varphi \rightarrow \mathcal{R}_{\text{dom}(\Gamma)}$, where $\text{dom}(\Gamma) \subset \mathcal{P}_\varphi$ for which Γ is defined. We then denote the explicit mapping of the trajectory variable ρ to a trajectory v as $\Gamma[\rho \rightarrow v]$. Given (Π, Γ) where Π and Γ are the trace mapping as used in the definition of Synchronous semantics of HyperTWTL and trajectory mapping respectively, we use $(\Pi, \Gamma) + k$ as the k^{th} successor of (Π, Γ) , i.e. the k^{th} reached can be reached after k steps from (Π, Γ) . In defining the semantics of Asynchronous HyperTWTL, we employ the asynchronous assignment $\Pi : \mathcal{V}_\varphi \times \mathcal{P}_\varphi \rightarrow \mathbb{T} \times \mathbb{Z}_{\geq 0}$ which maps each pair of trace variable and trajectory variable, (π, ρ) , into an indexed trace. Given a trace mapping Π , a trace variable π , a trajectory variable ρ , a trace t , and a pointer n , we denote the assignment that coincides with Π for every pair except for (π, ρ) which is mapped to (t, n) as $\Pi[(\pi, \rho) \rightarrow (t, n)]$. By $\Pi(\pi, \rho) = (t, p)$, we mean the event from the timed trace t at the position p is currently used in the analysis of trace and trajectory, π and ρ , respectively.

Let us recall that the hold operator $\mathbf{H}^d a_{\pi, \rho}$ states that the proposition a is

to be repeated for d time units in trace π and trajectory ρ . Similarly $\mathbf{H}^d \neg a_{\pi, \rho}$, requires that for d time units the proposition a should not be repeated in trace π and trajectory ρ . The trace set \mathbb{T} satisfies both sub-formulae in $\phi = \psi_1 \wedge \psi_2$ while in $\neg \psi$, \mathbb{T} , does not satisfy the given formula. A given formula with a concatenation operator in the form $\psi_1 \odot \psi_2$ specifies that every $t \in \mathbb{T}$ should satisfy ϕ_1 first and then immediately ϕ_2 must also be satisfied with one-time unit difference between the end of execution of ϕ_1 and the start of execution of ϕ_2 . The intended meaning of $[\phi]^{I, J}$ where $I = [\tau, \tau']$ and $J = [x, y]$ is the trace set \mathbb{T} must satisfy ϕ within the time window $[\tau, \tau']$ while the difference in time elapse between any pair of traces in the set \mathbb{T} must be between $[x, y]$.

For any given HyperTWTL formula φ we denote Δ as a map from $\mathcal{V}_\varphi \rightarrow \mathbb{Z}_{\geq 0}$ returns the time duration for each π in $dom(\Delta)$. We say $\Delta \in [\tau, \tau']$ whenever for all $\pi \in dom(\Delta)$, $\Delta(\pi) \in [\tau, \tau']$. Similarly, we say $\Delta \in [x, y]$ whenever for all distinct $\pi, \pi' \in dom(\Delta)$, $|\Delta(\pi') - \Delta(\pi)| \in [x, y]$. Given two indexed trace assignments Π and Π' defined within the same domain $dom(\Pi) = dom(\Pi')$, we denote $\Delta(\Pi, \Pi')(\pi)$ as the map from $\mathcal{V}_\varphi \rightarrow \mathbb{Z}_{\geq 0}$ that returns the time duration for each trace assignment as $\Delta(\Pi, \Pi')(\pi) = (\Pi'(\pi)).\tau - (\Pi(\pi)).\tau$. Likewise, given two distinct indexed trace and trajectory assignments (Π, Γ) and (Π', Γ') of the same domain, we denote $\Delta^j((\Pi, \Gamma), (\Pi', \Gamma'))$ as the duration of time that elapses from the current evaluation instant to the evaluation instance obtained after j steps. This is defined formally as $\Delta^j((\Pi, \Gamma), (\Pi', \Gamma')) = \Delta(\Pi, \Pi')(\pi)$, where $(\Pi', \Gamma') = (\Pi, \Gamma)^j$. Now, we denote the satisfaction of asynchronous semantics of HyperTWTL formula φ over trace mapping Π , trajectory mapping Γ , and a set of traces \mathbb{T} as $(\mathbb{T}, \Pi, \Gamma) \models_a \varphi$. The asynchronous semantics of HyperTWTL is presented in Table 2.

Table 2. Asynchronous semantics of HyperTWTL

$(\mathbb{T}, \Pi, \Gamma) \models_a \exists \pi. \varphi$	iff $\exists t \in \mathbb{T} \cdot (\mathbb{T}, \Pi[(\pi, \rho) \rightarrow (t, 0)], \Gamma) \models_a \varphi$ for all ρ
$(\mathbb{T}, \Pi, \Gamma) \models_a \forall \pi. \varphi$	iff $\forall t \in \mathbb{T} \cdot (\mathbb{T}, \Pi[(\pi, \rho) \rightarrow (t, 0)], \Gamma) \models_a \varphi$ for all ρ
$(\mathbb{T}, \Pi, \Gamma) \models_a \mathbf{E}\rho. \varphi$	iff $\exists v \in \mathcal{R}_{range(\Gamma)} : (\mathbb{T}, \Pi, \Gamma[\rho \rightarrow v]) \models_a \varphi$
$(\mathbb{T}, \Pi, \Gamma) \models_a \mathbf{A}\rho. \varphi$	iff $\forall v \in \mathcal{R}_{range(\Gamma)} : (\mathbb{T}, \Pi, \Gamma[\rho \rightarrow v]) \models_a \varphi$
$(\mathbb{T}, \Pi, \Gamma) \models_a \mathbf{H}^d a_{\pi, \rho}$	iff $a \in t[i].e$ for $(t, p) = \Pi(\pi, \rho), \forall p \in \{i, \dots, i + d\} \wedge (t[i + n].\tau - t[i].\tau) \geq d$, for some $n > 0$ and $i < d$
$(\mathbb{T}, \Pi, \Gamma) \models_a \mathbf{H}^d \neg a_{\pi, \rho}$	iff $a \notin t[i].e$ for $(t, p) = \Pi(\pi, \rho), \forall p \in \{i, \dots, i + d\} \wedge (t[i + n].\tau - t[i].\tau) \geq d$, for some $n > 0$ and $i < d$
$(\mathbb{T}, \Pi, \Gamma) \models_a \psi_1 \wedge \psi_2$	iff $((\mathbb{T}, \Pi, \Gamma) \models_a \psi_1) \wedge ((\mathbb{T}, \Pi, \Gamma) \models_a \psi_2)$
$(\mathbb{T}, \Pi, \Gamma) \models_a \neg \psi$	iff $\neg((\mathbb{T}, \Pi, \Gamma) \models_a \psi)$
$(\mathbb{T}, \Pi, \Gamma) \models_a \psi_1 \odot \psi_2$	iff $\exists i, j, k$ s.t. $i \leq k \leq j$ and $k = \min k' \mid i \leq k' \leq j$, $((\mathbb{T}_{[i, k]}, \Pi, \Gamma) \models_a \psi_1) \wedge ((\mathbb{T}_{[k+1, j]}, \Pi, \Gamma) \models_a \psi_2)$ for some $i, j \geq 0$
$(\mathbb{T}, \Pi, \Gamma) \models_a [\psi]^{[\tau, \tau'], [x, y]}$	iff $\exists i, j, k$ s.t. $k \geq i + \tau$, $(\mathbb{T}_{[k, i+\tau']}, \Pi, \Gamma) \models_a \psi \wedge \Delta((\Pi + j) - \Pi) \in [\tau, \tau'] \wedge \Delta^j((\Pi, \Gamma), (\Pi', \Gamma')) \in [x, y]$, for some $i, j > 0$

3 SMT-Based Model Checking for HyperTWTL

Given a collection of TKS \mathcal{M} , a HyperTWTL formula φ , and an unrolling bound $\|\varphi\|$ (discussed in the next section), the model checking problem is to determine whether $\mathcal{M} \models \varphi$. We assume that the input formula φ has been converted into a negation-normal form (NNF) denoted as $\neg\varphi$. The model checking approach takes as an input NNF of the HyperTWTL formula $\neg\varphi$ and TKS \mathcal{M} . Let us recall from Sect. 2 that φ can be either a synchronous or an asynchronous HyperTWTL formula. In the latter case, we need to translate the asynchronous HyperTWTL formula to an equivalent synchronous HyperTWTL formula. To achieve this, we first generate a set of invariant traces $inv(\mathbb{T})$ from a trace set \mathbb{T} generated over the TKS \mathcal{M} . We then construct an equivalent synchronous formula φ_s from the asynchronous formula φ such that $\mathbb{T} \models_a \varphi$ if and only if $inv(\mathbb{T}) \models_s \varphi_s$. For more details on this approach of converting an asynchronous HyperTWTL formula to an equivalent synchronous HyperTWTL formula, we refer the readers to the Appendix. Next, the TKS \mathcal{M} and NNF of the HyperTWTL formula $\neg\varphi$ are fed into an SMT encoder to generate a first-order logic formula of the form $\llbracket \mathcal{M}, \neg\varphi \rrbracket_{\|\varphi\|}$ by encoding the initial condition, the transition relations, and unrolling \mathcal{M} and $\neg\varphi$ to a depth of $\|\varphi\|$. Finally, we utilize off-the-shelf SMT solvers to solve the first order logic formula $\llbracket \mathcal{M}, \neg\varphi \rrbracket_{\|\varphi\|}$ and determine if $\mathcal{M} \models \neg\varphi$. If the SMT returns true, then a counterexample has been identified, otherwise, $\mathcal{M} \models \varphi$ holds.

3.1 Calculating Unrolling Bound from HyperTWTL

The satisfaction of a HyperTWTL formula can be decided within a fixed time bound. Let $\|\varphi\|$ denote the maximum time needed to satisfy the HyperTWTL formula φ and it can be computed as follows:

$$\|\varphi\| = \begin{cases} \|\varphi\| & \text{if } \varphi \in \{\exists\pi \cdot \varphi, \forall\pi \cdot \varphi\} \\ d & \text{if } \varphi \in \{\mathbf{H}^d a_\pi, \mathbf{H}^d \neg a_\pi\} \\ \max(\|\phi_1\|, \|\phi_2\|) & \text{if } \varphi \in \{\phi_1 \wedge \phi_2, \phi_1 \vee \phi_2\} \\ \|\phi\| & \text{if } \varphi = \neg\phi \\ \|\phi_1\| + \|\phi_2\| + 1 & \text{if } \varphi = \phi_1 \odot \phi_2 \\ \tau' & \text{if } \varphi \in \{\phi_1^{[\tau, \tau']}\} \end{cases} \quad (1)$$

We use the computed deadline $\|\varphi\|$ as the *unrolling bound* to determine the satisfiability of a HyperTWTL formula. Note, this contrasts the traditional BMC techniques which uses a given arbitrary unrolling bound.

Example 1. Let us consider a HyperTWTL formula φ as follows.

$$\varphi_1 = \forall\pi_1 \exists\pi_2 \cdot [\mathbf{H}^2 a_{\pi_1} \wedge \mathbf{H}^2 a_{\pi_2}]^{[0,2]} \odot [\mathbf{H}^2 a_{\pi_1} \vee \mathbf{H}^2 b_{\pi_2}]^{[3,7]} \quad (2)$$

Using Eq. (1), we can calculate $\|\varphi_1\| = 10$ time units.

3.2 Encoding the TKS

The encoding of a collection of TKS \mathcal{M} upto bound $\|\varphi\|$ into a first-order logic formula is inspired by the BMC encoding of LTL [3]. Intuitively, the states of the \mathcal{M} are represented by a set of variables S . Let S_i be new copies of S , where $i \in [0, \|\varphi\|]$ which captures the evolution of states over time. Consider the HyperTWTL formula φ_1 in Eq. (2) above, we use two identical copies of a given TKS to represent different paths π_1 and π_2 on the TKS, denoted as M_1 and M_2 , i.e. $\mathcal{M} = \langle M_1, M_2 \rangle$. Therefore, for each copy M_i , we unroll the transition relation $\llbracket M_i \rrbracket_{\|\varphi_1\|}$ as follows.

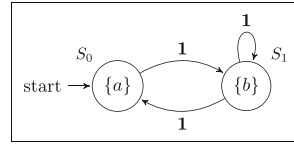


Fig. 2. TKS M

$$\llbracket M_i \rrbracket_{\|\varphi_1\|} = I(S_0) \wedge \bigwedge_{i=0}^{\|\varphi_1\|-1} R(S_i, S_{i+1}) \tag{3}$$

In Eq. (3), $I(S_0)$ is the characteristic function that encodes the initial states and $R(S_i, S_{i+1})$ is the function that encodes transition relation for states in S_i and their successor states in S_{i+1} between time steps i and $i + 1$.

Example 2. Consider the Kripke structure in Fig. 2 and a HyperTWTL formula

$$\varphi_2 = \forall \pi_1 \forall \pi_2 \cdot [\mathbf{H}^3 a_{\pi_1} \wedge \mathbf{H}^3 b_{\pi_2}]^{[0,3]} \tag{4}$$

For a bound $\|\varphi_2\| = 3$, we unroll the transition relation for copy M_1 as follows.

$$\llbracket M_1 \rrbracket_{\|3\|} = I(S_0) \wedge R(S_0, S_1) \wedge R(S_1, S_2) \wedge R(S_2, S_3) \tag{5}$$

3.3 Encoding the Inner TWTL Formula

Let φ be a HyperTWTL formula of the form $\varphi = Q_1 \pi_1 \dots Q_n \pi_n \cdot \phi$ where each $Q_j \in \{\forall, \exists\}$ ($j \in [1, n]$) and ϕ is the inner TWTL formula. For each $j \in [1, n]$, the path quantification $Q_j \pi_j$ is represented by

$$[Q_j \pi_j] = Q_j S_0 Q_j S_1 \dots Q_j S_{\|\varphi\|-1} \tag{6}$$

Given the negated formula $\neg \varphi$, we unroll the TWTL formula on a path π , with bound $\|\varphi\|$ resulting in a first-order logic formula which can be inductively defined in Table 3.

Example 3. Consider the HyperTWTL formula φ_2 in Eq. (4) above. The negation of Eq. (4) (refer Theorem 1) can be expressed as follows.

$$\neg \varphi_2 = \exists \pi_1 \exists \pi_2 \cdot \underbrace{[\mathbf{H}^3 \neg a_{\pi_1} \vee \mathbf{H}^3 \neg b_{\pi_2}]^{[0,3]}}_{\neg \phi} \tag{7}$$

Table 3. Encoding the inner TWTL formula

$\llbracket \mathbf{H}^d a_\pi \rrbracket_{i, \ \varphi\ }$	$:= \llbracket \mathbf{H}^d a_\pi \rrbracket_i \forall i \leq \ \varphi\ $
$\llbracket \mathbf{H}^d \neg a_\pi \rrbracket_{i, \ \varphi\ }$	$:= \llbracket \mathbf{H}^d \neg a_\pi \rrbracket_i \forall i \leq \ \varphi\ $
$\llbracket \phi_1 \wedge \phi_2 \rrbracket_{i, \ \varphi\ }$	$:= \llbracket \phi_1 \rrbracket_{i, \ \varphi\ } \wedge \llbracket \phi_2 \rrbracket_{i, \ \varphi\ }, \forall i \leq \ \varphi\ $
$\llbracket \neg \phi \rrbracket_{i, \ \varphi\ }$	$:= \neg \llbracket \phi \rrbracket_{i, \ \varphi\ }, \forall i \leq \ \varphi\ $
$\llbracket \phi_1 \odot \phi_2 \rrbracket_{i, \ \varphi\ }$	$:= \exists k = \arg \min_{i \leq k \leq \ \varphi\ } \llbracket \phi_1 \rrbracket_{i, k} \wedge$ $\llbracket \phi_2 \rrbracket_{k+1, \ \varphi\ } \forall i \leq \ \varphi\ $
$\llbracket [\phi]^{\lceil \tau, \tau' \rceil} \rrbracket_{i, \ \varphi\ }$	$:= \exists k \geq i + \tau, s.t. \llbracket \phi \rrbracket_{k, i+\tau} \wedge (\ \varphi\ - i \geq \tau'),$ $\forall i \leq \ \varphi\ $

From the structure of Eq. (7), the inner TWTL formula $\neg\phi$ is given as $\neg\phi = [\mathbf{H}^3 \neg a_{\pi_1} \vee \mathbf{H}^3 \neg b_{\pi_2}]^{[0,3]}$. Based on Table 3, unrolling $\neg\phi$ with a computed bound $\|\neg\varphi_2\| = 3$ can be expressed as follows.

$$\begin{aligned} \llbracket \neg\phi \rrbracket_{0, [3]} &= [\mathbf{H}^3 \neg a_{\pi_1} \vee \mathbf{H}^3 \neg b_{\pi_2}]_0^{[0,3]} \wedge [\mathbf{H}^3 \neg a_{\pi_1} \vee \mathbf{H}^3 \neg b_{\pi_2}]_1^{[0,3]} \\ &\quad \wedge [\mathbf{H}^3 \neg a_{\pi_1} \vee \mathbf{H}^3 \neg b_{\pi_2}]_2^{[0,3]} \wedge [\mathbf{H}^3 \neg a_{\pi_1} \vee \mathbf{H}^3 \neg b_{\pi_2}]_3^{[0,3]} \end{aligned} \quad (8)$$

3.4 Combining the Encodings

Given a HyperTWTL formula of the form $\varphi = Q_1 \pi_1 \dots Q_n \pi_n \cdot \phi$ and a collection of TKS $\mathcal{M} = \langle M_1, \dots, M_n \rangle$, the verification problem of HyperTWTL specifications can be formulated by constructing the first-order logic formula $\llbracket \mathcal{M}, \neg\varphi \rrbracket_{\|\varphi\|}$ as follows.

$$\llbracket \mathcal{M}, \neg\varphi \rrbracket_{\|\varphi\|} = [Q_1 \pi_1] \dots [Q_n \pi_n] \cdot \llbracket M_1 \rrbracket_{\|\varphi\|} \square_1 \dots \llbracket M_n \rrbracket_{\|\varphi\|} \square_n \llbracket \neg\phi \rrbracket_{0, \|\varphi\|} \quad (9)$$

where $[Q_j \pi_j]$ for $j \in [1, n]$ is defined in (6), $\llbracket M_j \rrbracket_{\|\varphi\|}$ for $j \in [1, n]$ is defined in (3), $\square_i = \wedge$ if $Q_i = \exists$, and $\square_i = \rightarrow$ if $Q_i = \forall$, for $i \in \mathcal{V}_\varphi$ and $\neg\phi$ is the negated inner TWTL formula ϕ of the HyperTWTL formula φ .

Example 4. Let us consider the Kripke structure in Fig. 1 and the HyperTWTL formula $\varphi = \forall \pi_1 \forall \pi_2 \cdot [\mathbf{H}^3 a_{\pi_1} \wedge \mathbf{H}^3 b_{\pi_2}]^{[0,3]}$ with $\|\varphi\| = 3$. Let $\mathcal{M} = \langle M_1, M_2 \rangle$ denote identical collection of the Kripke structure representing paths π_1 and π_2 respectively. The resulting combined first-order logic formula to be solved is given as follows.

$$\llbracket \mathcal{M}, \neg\varphi \rrbracket_3 = [\exists_1 \pi_1] \cdot [\exists_2 \pi_2] \cdot \llbracket M_1 \rrbracket_3 \wedge \llbracket M_2 \rrbracket_3 \wedge \llbracket \neg\phi \rrbracket_{0,3} \quad (10)$$

Theorem 1. Given a collection TKS \mathcal{M} , a HyperTWTL formula φ with an unrolling bound of $\|\varphi\|$ and sets of traces \mathbb{T} over \mathcal{M} , if $\llbracket \mathcal{M}, \neg\varphi \rrbracket_{\|\varphi\|}$ is satisfiable, i.e. $(\mathbb{T}, \Pi) \not\models_s \varphi$, then $\mathcal{M} \not\models_s \varphi$.

4 Experimental Results

To demonstrate the effectiveness of our approach, we consider two case studies and compare their performance with the automata-based HyperTWTL model checking approach [9]. We present the details of these case studies and the obtained results in the following sections.

4.1 Case Study I: Autonomous Security Robots

Our Case Study-1 resembles a security patrol within a community with multiple autonomous security robots [24]. In this case study, the autonomous security robots are augmented with intelligent video surveillance systems that move along patrol routes to different areas while identifying potential intruders, incidents, crimes, etc., and relaying information to an operator in a remote base station for data processing. Let us consider an environment to be patrolled in Fig. 3 which is composed of 2 initial positions I_1 and I_2 , 2 charging stations C_1 and C_2 , 12 allowable states P_1, \dots, P_{12} and 4 regions of interest to be patrolled R_1 to R_4 . Each patrol starts from any of the initial states (grey) and subsequently proceeds along the patrol routes through the allowable states (white). On each patrol, it is required each security robot surveils all regions of interest (blue) before proceeding to any of the charging stations (yellow). We abstract the patrol environment into a weighted graph where the nodes represent the initial states, charging stations, allowable states, and regions of interest, while the edges represent transitions between the nodes, and the assigned weights represent travel times associated with the transitions. We further abstract the motion of each security robot into a transition system derived from the patrol environment by splitting all transitions to have an edge weight of 1 time unit. Based on this case study,

Table 4. Requirements expressed in HyperTWTL in Case Study I

No.	Description	HyperTWTL Specification
1	Mutation Testing	$\varphi_1 = \exists \pi_1 \forall \pi_2 \cdot [\mathbf{H}^d t_{\pi_1}^m \wedge \mathbf{H}^d t_{\pi_2}^{-m}]^{[0, T_9]} \wedge [\mathbf{H}^1 I_{\pi_1} = \mathbf{H}^1 I_{\pi_2}]^{[0, T_1]} \odot [\mathbf{H}^1 R_{1\pi_1} \wedge \mathbf{H}^1 R_{1\pi_2}]^{[T_2, T_3]} \odot [\mathbf{H}^1 R_{2\pi_1} \wedge \mathbf{H}^1 R_{2\pi_1}]^{[T_4, T_5]} \odot [\mathbf{H}^1 R_{3\pi_2} \wedge \mathbf{H}^1 R_{3\pi_2}]^{[T_6, T_7]} \odot [\mathbf{H}^1 R_{4\pi_1} \wedge \mathbf{H}^1 R_{4\pi_2}]^{[T_8, T_9]} \odot [\mathbf{H}^1 C_{\pi_1} \neq \mathbf{H}^1 C_{\pi_2}]^{[T_{10}, T_{11}]}$, where $d = T_9$
2	Opacity	$\varphi_2 = \exists \pi_1 \exists \pi_2 \cdot [\mathbf{H}^1 I_{\pi_1} \wedge \mathbf{H}^1 I_{\pi_2}]^{[0, T_1]} \odot [\mathbf{H}^1 R_{1\pi_1} \wedge \mathbf{H}^1 R_{1\pi_2}]^{[T_2, T_3]} \odot [\mathbf{H}^1 R_{2\pi_1} \wedge \mathbf{H}^1 R_{2\pi_1}]^{[T_4, T_5]} \odot [\mathbf{H}^1 R_{3\pi_2} \wedge \mathbf{H}^1 R_{3\pi_2}]^{[T_6, T_7]} \odot [\mathbf{H}^1 R_{4\pi_1} \wedge \mathbf{H}^1 R_{4\pi_2}]^{[T_8, T_9]} \odot [\mathbf{H}^1 C_{\pi_1} \wedge \mathbf{H}^1 C_{\pi_2}]^{[T_{10}, T_{11}]}$
3	Side-Channel Timing Attacks	$\varphi_3 = \forall \pi_1 \forall \pi_2 \cdot \mathbf{A} \rho \mathbf{E} \rho' \cdot [\mathbf{H}^1 I_{\pi_1, \rho} \wedge \mathbf{H}^1 I_{\pi_2, \rho'}]^{[0, T_1]} \rightarrow [\mathbf{H}^1 R_{1\pi_1, \rho} \wedge \mathbf{H}^1 R_{1\pi_2, \rho'}]^{[T_2, T_3]} \odot [\mathbf{H}^1 R_{2\pi_1, \rho} \wedge \mathbf{H}^1 R_{2\pi_1, \rho'}]^{[T_4, T_5]} \odot [\mathbf{H}^1 R_{3\pi_2, \rho'} \wedge \mathbf{H}^1 R_{3\pi_2, \rho'}]^{[T_6, T_7]} \odot [\mathbf{H}^1 R_{4\pi_1, \rho} \wedge \mathbf{H}^1 R_{4\pi_2, \rho'}]^{[T_8, T_9]} \odot [\mathbf{H}^1 C_{\pi_1, \rho} \wedge \mathbf{H}^1 C_{\pi_2, \rho'}]^{[T_{10}, T_{11}]}$
4	Non-Interference	$\varphi_4 = \forall \pi_1 \exists \pi_2 \cdot \mathbf{A} \rho \cdot [\mathbf{H}^1 I_{\pi_1, \rho} \neq \mathbf{H}^1 I_{\pi_2, \rho}]^{[0, T_1]} \rightarrow [\mathbf{H}^1 R_{1\pi_1, \rho} \wedge \mathbf{H}^1 R_{1\pi_2, \rho}]^{[T_2, T_3]} \odot [\mathbf{H}^1 R_{2\pi_1, \rho} \wedge \mathbf{H}^1 R_{2\pi_1, \rho}]^{[T_4, T_5]} \odot [\mathbf{H}^1 R_{3\pi_2, \rho} \wedge \mathbf{H}^1 R_{3\pi_2, \rho}]^{[T_6, T_7]} \odot [\mathbf{H}^1 R_{4\pi_1, \rho} \wedge \mathbf{H}^1 R_{4\pi_2, \rho}]^{[T_8, T_9]} \odot [\mathbf{H}^1 C_{\pi_1, \rho} = \mathbf{H}^1 C_{\pi_2, \rho}]^{[T_{10}, T_{11}]}$

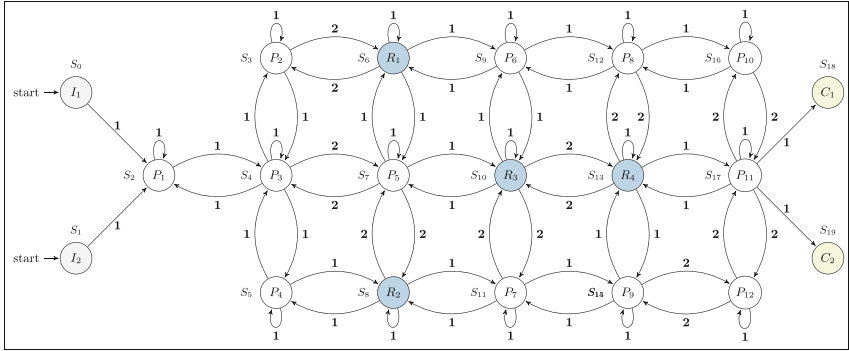


Fig. 3. The Patrol environment

we consider 4 different scenarios with 4 different HyperTWTL specifications, including mutation testing, opacity, side-channel attacks and non-interference. We formalize these requirements in HyperTWTL as follows. Note, in φ_2 and all subsequent formulae, “=” is not an arithmetic operator but a notation of simplification such that $[\mathbf{H}^1 I_{\pi_1} = \mathbf{H}^1 I_{\pi_2}]$ stands for $\bigwedge_{i \in I} ([\mathbf{H}^1 i_{\pi_1} \wedge \mathbf{H}^1 i_{\pi_2}])$.

Requirement 1 (Mutation Testing): An interesting application of hyper-property is the efficient generation of test cases for mutation testing. Let us assume that traces from all robots within the surveillance system are labeled as either *mutated* (t^m) or *non-mutated* (t^{-m}). We map t^m to π_1 and all other non-mutated traces t^{-m} to π_2 . This requirement guarantees that even if π_2 starts from the same initial state (I_1 or I_2) as π_1 , they eventually proceed to different charging states (C_1 or C_2). This can be formalized as a synchronous HyperTWTL formula φ_1 as shown in Table 4.

Requirement 2 (Opacity): Information-flow security policies define what users can learn about a system while (partially) observing the system. A system is said to be opaque if it meets two requirements: (i) there exist at least two executions of the system mapped to π_1 and π_2 with the same observations but bearing a distinct secret, and (ii) the secret of each path cannot be accurately determined only by observing the system. For example, let the surveillance route be secret, and the initial state I_1 or I_2 be the only information a system user can observe. This can be formalized as a synchronous HyperTWTL formula φ_2 as shown in Table 4.

Requirement 3 (Side-Channel Timing Attacks): A side-channel timing attack is a security threat that attempts to acquire sensitive information from the surveillance mission by exploiting the execution time of the mission. Let us assume two security robots start from any initial states I_1 or I_2 simultaneously, and their executions are mapped to π_1 and π_2 , respectively. To design a countermeasure against this attack, it is required that for any pair of executions, if

Table 5. Verification results of HyperTWTL properties for Case Study I

HyperTWTL Req.	Description	Verdict	Z3		CVC4		AMC	
			Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)
φ_1	Mutation Testing	SAT	1.450	8.604	0.844	10.554	16.103	16.893
φ_2	Opacity	UNSAT	1.453	8.691	0.892	10.564	15.952	17.110
φ_3	Side-Channel Timing Attacks	SAT	1.427	8.613	0.839	10.415	16.186	16.950
φ_4	Non-Interference	SAT	1.421	8.611	0.885	10.425	16.225	16.832

Table 6. Verification time for HyperTWTL properties for Case Study I

HyperTWTL properties	Z3 (s)				CVC4 (s)				AMC (s)			
	Unrolling bounds ($\ \varphi\ $)				Unrolling bounds ($\ \varphi\ $)				Unrolling bounds ($\ \varphi\ $)			
	51	75	100	125	51	75	100	125	51	75	100	125
φ_1	1.450	2.588	3.629	4.741	0.844	1.775	2.772	3.682	16.103	17.165	18.386	19.507
φ_2	1.453	2.541	3.652	4.782	0.892	1.744	2.628	3.671	15.952	16.285	17.733	18.895
φ_3	1.427	2.573	3.681	4.794	0.839	1.710	2.631	3.766	16.189	17.085	18.386	19.297
φ_4	1.421	2.595	3.648	4.766	0.885	1.725	2.711	3.729	16.225	17.198	18.738	19.098

both robots start from any initial states simultaneously, they should reach the charging state C_1 or C_2 within close enough time after finishing their surveillance tasks. This can be formalized as an asynchronous HyperTWTL formula φ_3 as shown in Table 4.

Requirement 4 (Non-interference): Non-interference is a security policy that seeks to restrict the flow of information within a system. This policy requires that low-security variables be independent of high-security variables, i.e., one should not be able to infer information about a high-security variable by observing low-security variables. For a set of traces, let us assume that the initial state I_1 or I_2 is a high variable (high security) and paths from initial states to charging states C_1 or C_2 through R_1, \dots, R_6 denote low variable (low security). The surveillance system satisfies non-interference if, for all executions, there exists another execution that starts from a different high variable (i.e., the initial states are different), and at the end of the mission, they are in the same low variable states (i.e., charging states C_1 or C_2 are the same). This can be formalized as an asynchronous HyperTWTL formula φ_4 as shown in Table 4.

Table 7. Memory consumption for HyperTWTL verification for Case Study I

HyperTWTL properties	Z3 (MB)				CVC4 (MB)				AMC (MB)			
	Unrolling bounds ($\ \varphi\ $)				Unrolling bounds ($\ \varphi\ $)				Unrolling bounds ($\ \varphi\ $)			
	51	75	100	125	51	75	100	125	51	75	100	125
φ_1	8.604	13.965	20.091	24.214	10.554	16.117	23.343	27.868	17.933	19.573	26.304	32.830
φ_2	8.691	13.883	20.082	24.253	10.564	16.065	22.957	27.748	16.057	18.463	25.457	31.487
φ_3	8.613	13.834	20.051	24.269	10.415	16.219	23.117	27.445	17.578	19.931	26.647	31.608
φ_4	8.611	13.840	20.067	24.283	10.425	16.269	23.002	27.678	17.711	19.156	26.483	31.372

4.2 Case Study 1: Experimental Results

The conversion from the TKS and the HyperTWTL specifications to first-order logic expressions (resembling Eq. (9)) is implemented in Python 3.7. The obtained first-order logic formula is then fed to Z3 and CVC4 SMT solvers for verification on a Windows 10 system with 64 GB RAM and Intel Core(TM) i9-10900 CPU (3.70 GHz). Z3 and CVC4 are widely known for their industrial applications [4, 29]. The following time bounds are considered for the verification of all the HyperTWTL properties in Table 4: $T_1 = 1$, $T_2 = 2$, $T_3 = 3$, $T_4 = 4$, $T_5 = 7$, $T_6 = 8$, $T_7 = 9$, $T_8 = 10$, $T_9 = 12$, $T_{10} = 13$ and $T_{13} = 15$. Since the time bounds are the same for HyperTWTL formulae from φ_1 – φ_4 , their unrolling bound is also the same, i.e., $\|\varphi\| = 51$.

In the first set of experiments, we verify the HyperTWTL specifications using the Z3 and CVC4 SMT solvers and compare them with the automata-based model checking (AMC) approach in [9]. The obtained results are shown in Table 5. Note, since φ_3 and φ_4 are asynchronous HyperTWTL specifications, we convert them to equivalent synchronous HyperTWTL specifications following the method described in [9] before running these experiments. We observe that specifications φ_1 , φ_3 and φ_4 were satisfied using both Z3 and CVC4 as well as the AMC approach, whereas φ_2 was unsatisfied. We also observe that the verification time of the HyperTWTL formulae never exceeded 1.453 seconds in Z3 and 0.892 seconds in CVC4. In contrast, it took up to 16.225 seconds in the AMC approach to verify these properties. This shows a $11\times$ and $19\times$ speed up for Z3 and CVC4, respectively, regarding execution time. Regarding the memory consumption, verifying these specifications using Z3 and CVC4 never exceeded 8.691 MB and 10.564 MB, respectively. In contrast, AMC consumed up to 17.110 MB of memory. This shows that our SMT-based verification approach is $2\times$ and $1.6\times$ more memory-efficient while using Z3 and CVC4, respectively.

In the second set of experiments, we evaluate the performance of Z3 and CVC4 solvers for verifying HyperTWTL properties against different unrolling bounds, i.e., analyze the impact of $\|\varphi\|$ on the verification performance. For this, we vary the $\|\varphi\|$ in the range of 51 to 125 for φ_1 – φ_4 and record their respective verification time and memory as shown in Tables 6 and 7. Table 6 shows that CVC4 is faster than Z3 and the AMC approach regarding verification time. For instance, while verifying φ_1 for $\|\varphi\| = 125$, Z3 and AMC approach take 4.741 seconds and 19.507 seconds respectively. In contrast, verifying the same property for $\|\varphi\| = 125$ using CVC4 takes only 3.682 seconds. This shows that CVC4 is faster than both the Z3 and AMC approaches. Similar pattern is observed while verifying φ_4 for $\|\varphi\| = 75$. Z3 and AMC approaches take 2.595 seconds and 18.738 seconds, respectively, whereas CVC4 takes only 1.724 seconds. A similar trend is also observed for the rest of the HyperTWTL properties. We also observe that execution time increases linearly for verifying HyperTWTL properties against increasing unrolling bounds, irrespective of the techniques used for verification.

Consequently, as shown in Table 7, we observe that Z3 consumes less memory than CVC4 and the AMC for verifying the HyperTWTL properties. For

Table 8. Requirements expressed in HyperTWTL in Case Study II

No.	Description	HyperTWTL Specification
5	Shortest Path	$\varphi_5 = \exists \pi_1 \forall \pi_2. [\mathbf{H}^1 S_{\pi_1} \wedge \mathbf{H}^1 S_{\pi_2}]^{[0, T_1]} \odot [\mathbf{H}^1 E_{1\pi_1} \wedge \mathbf{H}^1 E_{1\pi_2}]^{[T_2, T_3]} \odot [\mathbf{H}^1 E_{2\pi_1} \wedge \mathbf{H}^1 E_{2\pi_2}]^{[T_4, T_5]} \odot [\mathbf{H}^1 E_{3\pi_1} \wedge \mathbf{H}^1 E_{3\pi_2}]^{[T_6, T_7]} \odot [\mathbf{H}^1 E_{4\pi_1} \wedge \mathbf{H}^1 E_{4\pi_2}]^{[T_8, T_9]} \odot [\mathbf{H}^1 E_{5\pi_1} \wedge \mathbf{H}^1 E_{5\pi_2}]^{[T_{10}, T_{11}]} \odot [\mathbf{H}^1 E_{6\pi_1} \wedge \mathbf{H}^1 E_{6\pi_2}]^{[T_{12}, T_{13}]} \odot ([\mathbf{H}^1 L_{\pi_2}]^{[T_{14}, T_{15}]} \wedge [\mathbf{H}^1 L_{\pi_1}] \rightarrow [\mathbf{H}^1 L_{\pi_2}]^{[T_{14}, T_{15}]})$
6	Symmetry	$\varphi_6 = \exists \pi_1 \forall \pi_2. [\mathbf{H}^1 S_{\pi_1} \wedge \mathbf{H}^1 S_{\pi_2}]^{[0, T_1]} \odot [\mathbf{H}^1 E_{1\pi_1} \wedge \mathbf{H}^1 E_{1\pi_2}]^{[T_2, T_3]} \odot [\mathbf{H}^1 E_{2\pi_1} \wedge \mathbf{H}^1 E_{3\pi_1}]^{[T_4, T_7]} \odot [\mathbf{H}^1 E_{4\pi_1} \wedge \mathbf{H}^1 E_{5\pi_2}]^{[T_6, T_{11}]} \odot [\mathbf{H}^1 E_{6\pi_1} \wedge \mathbf{H}^1 E_{6\pi_2}]^{[T_{12}, T_{13}]} \odot [\mathbf{H}^1 L_{\pi_1} \wedge \mathbf{H}^1 L_{\pi_2}]^{[T_{14}, T_{15}]}$
7	Linearizability	$\varphi_7 = \forall \pi_1 \exists \pi_2. [\mathbf{H}^1 S_{\pi_1} = \mathbf{H}^1 S_{\pi_2}]^{[0, T_1]} \odot [\mathbf{H}^1 E_{1\pi_1} \wedge \mathbf{H}^1 E_{1\pi_2}]^{[T_2, T_3]} \odot [\mathbf{H}^1 E_{2\pi_1} \wedge \mathbf{H}^1 E_{2\pi_2}]^{[T_4, T_5]} \odot [\mathbf{H}^1 E_{3\pi_1} \wedge \mathbf{H}^1 E_{3\pi_2}]^{[T_6, T_7]} \odot [\mathbf{H}^1 E_{4\pi_1} \wedge \mathbf{H}^1 E_{4\pi_2}]^{[T_8, T_9]} \odot [\mathbf{H}^1 E_{5\pi_1} \wedge \mathbf{H}^1 E_{5\pi_2}]^{[T_{10}, T_{11}]} \odot [\mathbf{H}^1 E_{6\pi_1} \wedge \mathbf{H}^1 E_{6\pi_2}]^{[T_{12}, T_{13}]} \odot [\mathbf{H}^1 L_{\pi_1} = \mathbf{H}^1 L_{\pi_2}]^{[T_{14}, T_{15}]}$

instance, while verifying φ_2 for $\|\varphi\| = 100$, CVC4 and AMC consume 22.957 MB and 25.457 MB in memory, respectively. In contrast, verifying the same property for $\|\varphi\| = 100$ using Z3 takes 20.082 MB. This shows that Z3 is more memory efficient than CVC4 and, of course, AMC. Similarly, while verifying φ_4 for $\|\varphi\| = 51$, CVC4 and the AMC consume 10.425 MB and 17.711 MB in memory, respectively, whereas Z3 consumes only 8.611 MB. A similar trend of memory consumption is observed for the rest of the HyperTWTL specifications. Indeed, we also observe a linear trend in memory consumption with increasing HyperTWTL unrolling bound irrespective of the verification method used.

4.3 Case Study II: Industrial Inspection Robots

To further demonstrate the efficiency of the proposed verification algorithm, we consider case study II which resembles a real-world end-to-end robotic solution that automates industrial inspections [2]. In this case study, robots are used to monitor complex installations of energy and industrial processing plants to provide up-to-date and reliable data on plant machinery to enhance industrial operations. Plant oper-

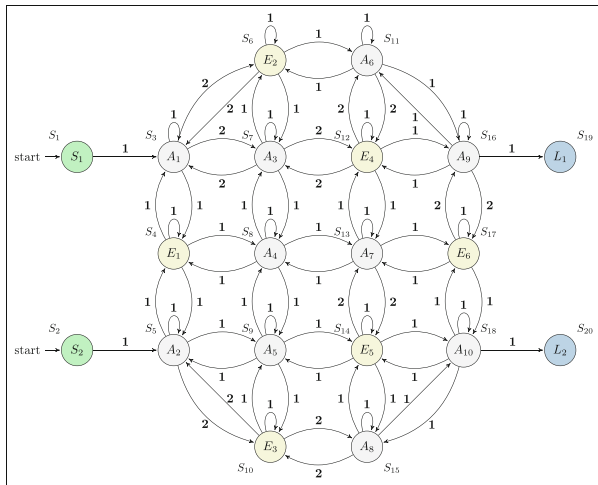


Fig. 4. The Inspection environment

Table 9. Verification results of HyperTWTL properties for Case Study II

HyperTWTL Req.	Description	Verdict	Z3		CVC4		AMC	
			Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)
φ_5	Shortest Path	SAT	5.112	10.423	4.587	12.401	18.143	19.860
φ_6	Symmetry	SAT	5.985	10.258	4.937	12.537	18.353	20.315
φ_7	Linearizability	UNSAT	5.674	10.695	4.185	12.118	19.058	21.865

ators use the collected data to maximize equipment uptime, enhance operations, and improve safety while reducing operations costs. Consider the floor of the plant to be routinely inspected in Fig. 4 which is composed of 2 initial positions S_1 and S_2 , 6 equipment to be routinely inspected E_1, \dots, E_6 , 2 landing stations L_1 and L_2 , and 10 allowable states A_1, \dots, A_{10} . On each inspection routine, the robot starts from any of the initial states (green), proceeds to collect data from installed equipment (yellow) on the plant floor, and then finally to the landing states (blue). We abstract the inspection environment into a weighted graph where the nodes represent the initial states, landing stations, allowable states, and installed equipment, while the edges represent transitions between the nodes, and the assigned weights represent travel times associated with the transitions. Based on this case study, we consider 3 different scenarios with 3 different HyperTWTL specifications that include optimality, symmetry, and linearizability.

Requirement 5 (Shortest Path): Optimality requirements are important hyperproperties in robotic applications. One such requirement is finding the shortest path over the human-robot collaboration environment. Let us consider a scenario where a robot starts the inspection from any of the initial states S_1 or S_2 , followed by an inspection and gathering data from equipment E_1, E_2, E_3, E_4, E_5 , and E_6 . After gathering data from all installed equipment, the robot finally proceeds to any of the landing states L_1 and L_2 . Given a set of executions, there exists an execution mapped to π_2 that reaches a landing state from the initial states before any other execution mapped to π_1 . This can be formalized as a synchronous HyperTWTL formula φ_5 as shown in Table 8.

Requirement 6 (Symmetry): Let us assume that two robots are available to inspect and gather data from equipment E_1, \dots, E_6 . In this case, one robot must inspect and gather data from equipment E_1, E_2, E_4 , and E_6 , while the other robot should inspect and gather data from equipment E_1, E_3, E_5 , and E_6 . We assume E_2 and E_4 are mapped to π_1 if and only if E_3 and E_5 are already mapped to π_2 and vice-versa. This can be formalized as a synchronous HyperTWTL formula φ_6 as shown in Table 8.

Table 10. Scalability Analysis for SMT-based Model Checking for HyperTWTL

TKS size	Unrolling bound $\ \varphi\ $	Z3		CVC4		AMC	
		Time (s)	Memory (MB)	Time (s)	Memory (MB)	Time (s)	Memory (MB)
20^2	51	1.45	8.60	0.84	10.33	16.10	17.93
20^4	100	115.46	25.81	73.035	42.10	692.20	151.62
20^6	150	1128.04	105.83	619.07	219.76	2145.82	802.73
20^8	200	4500.88	296.32	2721.56	532.91	–	–
20^{10}	250	11978.13	1074.41	8859.02	1933.84	–	–

Requirement 7 (Linearizability): The principle underlying linearizability is that the whole system operates as if executions from all human-robot collaborations are from one collaboration. Thus, linearizability is a correctness condition that guarantees consistency across concurrent executions of a given system. Any pair of traces must occupy the same states within the given mission time for the surveillance mission. At the same time, it is also essential to ensure that the mission’s primary goal to inspect and gather data from installed equipment is completed before proceeding to the landing states L_1 or L_2 is not violated. This can be formalized as a synchronous HyperTWTL formula φ_7 as in Table 8.

4.4 Case Study II: Experimental Results

All experiments are performed in the same computing environment and follow the same procedure as case study I. The following time bounds are considered for the verification of all the HyperTWTL properties in Table 8: $T_1 = 1$, $T_2 = 2$, $T_3 = 4$, $T_4 = 5$, $T_5 = 8$, $T_6 = 9$, $T_7 = 13$, $T_8 = 14$, $T_9 = 19$, $T_{10} = 20$, $T_{11} = 23$, $T_{12} = 24$, $T_{13} = 26$, $T_{14} = 27$, and $T_{15} = 29$. Since the time bounds are the same for HyperTWTL formulae from φ_5 – φ_7 , their unrolling bound is also the same, i.e., $\|\varphi\| = 129$. Similar to case study 1, we verify the HyperTWTL specification using Z3 and CVC4 SMT solvers and compare them with the automata-based model checking (AMC) approach in [9]. The obtained results for case study II are shown in Table 9. From Table 9, we observe that φ_5 and φ_6 were satisfied using Z3, CVC4 and AMC whereas φ_7 was unsatisfied. Once again, we observe that the verification time never exceeded 5.985 seconds in Z3, 4.937 seconds in CVC4, and 19.058 seconds in AMC. This shows a $3.55\times$ and $3.96\times$ speed up for Z3 and CVC4, respectively, regarding execution time. We also observe that while verifying the above specifications, the memory consumed never exceeded 10.695MB in Z3, 12.537 MB in CVC4, and 21.865 MB in AMC. Once again, this shows that our SMT-based verification approach is $2\times$ and $1.8\times$ more memory-efficient while using Z3 and CVC4, respectively.

5 Scalability Analysis

In our last set of experiments, we evaluate the scalability of our proposed verification approach by varying the size of TKS \mathcal{T} and verifying them using our

approach vs. the AMC approach. The size of the \mathcal{T} ranges from 20^2 to 20^{10} with randomly generated transitions. For this experiment, we consider φ_2 as the specification and vary the unrolling bound $\|\phi\|$ in the range of 50–250. All experiments are performed in the same computing environment as case studies I and II. The obtained results for the scalability analysis are presented in Table 10. Table 10 shows that for φ_2 , the verification time increases with the increasing size of the TKS. However, the results shown in Table 10 also suggest that our proposed verification approach using SMT solvers, i.e., Z3 and CVC4, are more scalable than the previously proposed AMC approach. For instance, for the TKS with 20^2 states and $\|\phi\| = 50$, it takes only 1.45 and 0.84 seconds for Z3 and CVC4, respectively for verification. However, verifying the same φ_2 using AMC takes 16.10. This shows approximately $11\times$ and $19\times$ speedup for our approach compared to the AMC. Similarly, for a TKS with 20^6 states and $\|\phi\| = 150$, Z3 and CVC4 takes 1128.04 seconds and 619.07 seconds, respectively to verify φ_2 . Verifying the same requirement using the AMC approach requires 2145.82 seconds. This shows approximately $2\times$ and $3.5\times$ speedup for our approach compared to the AMC. The comparison of memory consumption for verification also follows the same trend. Interestingly, while verifying the TKSs with 20^8 and 20^{10} states for $\|\phi\| = 200$ and $\|\phi\| = 250$, the AMC approach experienced a state-space explosion. In contrast, our proposed SMT approach successfully verified the property using Z3 and CVC4 in 4500.88 and 2721.56 seconds, respectively.

Bounded model checking with SMT has been successfully used in developing safety-critical industrial systems for decades [4, 29]. We believe that engineers can use our proposed HyperTWTL model checking approach to verify a wide range of safety and security properties of large-scale, complex, and safety-critical robotic missions.

6 Related Works

Model checking [11] has extensively been used to verify hyperproperties of models abstracted as transition systems by examining their related state transition graphs [10]. In [15], the first model checking algorithms for HyperLTL and HyperCTL* employing alternating automata were proposed, which was also adopted in [8, 21] to verify HyperMTL properties. An extensive study on the complexity of verifying hyperproperties with model checking is presented in [5]. The bounded model checking approach has recently become popular in the verification of HyperLTL specifications [16, 22, 23, 27]. Specifically, the work in [31] is most relevant to ours, where the authors use HyperLTL and an SMT solver for robotic mission planning. The work in [31] was indeed the first attempt to use hyperproperties for robotic mission planning. However, HyperLTL cannot express tasks with explicit time constraints, which motivates our contribution in this paper. Very recently, the authors in [9] proposed the HyperTWTL formalism and an automata-based model checking approach for verifying them. In contrast to [9], this paper presents a bounded model checking approach for verifying HyperTWTL specification using SMT solvers for enhanced verification performance regarding verification time and memory.

7 Conclusion

This paper introduced a bounded model checking approach for HyperTWTL using SMT solvers, contrasting the existing automata-based HyperTWTL verification. Specifically, we reduce the HyperTWTL model checking problem to a first-order logic satisfiability problem and then use two state-of-the-art SMT solvers, i.e., Z3 and CVC4, for verification. Using two case studies, Technical Surveillance Squadron (TESS) and Robotic Industrial Inspection, and a scalability study, we showed that the proposed bounded model checking approach can efficiently verify HyperTWTL properties compared to the AMC approach while offering up to $19\times$ speed up in terms of verification time and up to $2\times$ memory efficiency. Our scalability analysis results also show that our proposed approach can verify large systems, whereas the previously reported automata-based HyperTWTL verification method suffers from a state-space explosion.

Appendix

Asynchronous HyperTWTL to Synchronous HyperTWTL

The process to convert a given asynchronous HyperTWTL formula to a synchronous HyperTWTL formula has two parts. First, we generate *invariant* set of traces $inv(\mathbb{T})$ for the corresponding trace set \mathbb{T} generated over model \mathcal{T} . This allows for the synchronization of interleaving traces while reconciling the synchronous and asynchronous semantics of HyperTWTL. Secondly, we construct an equivalent synchronous formula $\hat{\varphi}$ from an asynchronous formula φ such that $\mathbb{T} \models_a \varphi$ if and only if $inv(\mathbb{T}) \models_s \hat{\varphi}$. These steps are described as follows.

Invariant Trace Generation. To construct an equivalent HyperTWTL synchronous formula $\hat{\varphi}$ from a given asynchronous HyperTWTL formula φ , we require that HyperTWTL be *stutter insensitive* [26]. To achieve this, we define the variable γ_π^ρ needed for the evaluation of the atomic propositions across traces. Thus, given a pair of traces π_1 and π_2 , γ_π^ρ ensures that all propositions in both traces exhibit the identical sequence at all timestamps. However, since timestamps proceed at different speeds in different traces such as π_1 and π_2 , a trajectory ρ is used to determine which trace moves and which trace stutters at any time point. In an attempt to synchronize traces once non-aligned timestamps are identified by a trajectory, silent events (ϵ) are introduced between the time stamps of the trace. For all $t \in \mathbb{T}$, we denote $inv(\mathbb{T})$ as the maximal set of traces defined over \mathcal{A}_ϵ where $\Sigma_\epsilon = \Sigma \cup \epsilon$. Consider a trace $t = (3, \{b\})(6, \{a\})(8, \{b\}) \dots$. The trace $t' \in inv(\mathbb{T})$ can be generated as $inv(t) = \epsilon\epsilon\epsilon b\epsilon\epsilon a\epsilon b \dots$. We now construct the synchronous HyperTWTL formula to reason about the trace set $inv(\mathbb{T})$.

Synchronous HyperTWTL Formula Construction. We now construct a synchronous formula $\hat{\varphi}$ that is equivalent to the asynchronous HyperTWTL φ . Intuitively, the asynchronous formula of HyperTWTL φ depends on a finite

interval of a timed trace. Thus, we can replace the asynchronous formula φ with a synchronous formula $\hat{\varphi}$ that encapsulates the interval patterns in the asynchronous formula φ . Given a bounded asynchronous formula φ , we define β_φ as the projected period required to satisfy the asynchronous formula. Inductively, β_φ can be defined as: $\beta_{\mathbf{H}^d a} = d$ for the \mathbf{H} operator; $\beta_{\varphi_1 \wedge \varphi_2} = \max(\beta_{\varphi_1}, \beta_{\varphi_2})$ for the \wedge operator; $\beta_{\neg\varphi} = \beta_\varphi$ for the \neg operator; $\beta_{\varphi_1 \odot \varphi_2} = \beta_{\varphi_1} + \beta_{\varphi_2} + 1$ for the \odot operator; $\beta_{[\varphi]^{x,y}} = \text{up}(X) + \text{up}(Y)$ for the $[\]$ operator, where $\text{up} \rightarrow \mathbb{Z}_{\geq 0}$ returns the upper bound of a predefined time bound. We then construct a synchronous formula $\hat{\varphi}$ from an asynchronous formula φ by replacing the time required for the satisfaction of φ with the appropriate ρ_φ .

References

1. Alpern, B., et al.: Defining liveness. *Inf. Process. Lett.* **21**(4), 181–185 (1985)
2. ANYbotics. Automation & Digitalization at Scale: ANYmal Makes the Case at BASF (2021). <https://www.anybotics.com/anymal-makes-the-case-at-basf-chemical-plant/>
3. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking (2003)
4. Bjørner, N.: Z3 and SMT in industrial R&D. In: FM, pp. 675–678. Springer (2018)
5. Bonakdarpour, B., et al.: The complexity of monitoring hyperproperties. In: 2018 IEEE 31st CSF, pp. 162–174. IEEE (2018)
6. Bonakdarpour, B., et al.: Monitoring hyperproperties by combining static analysis and runtime verification. In: ISoLA, pp. 8–27. Springer (2018)
7. Bonakdarpour, B., et al.: Controller synthesis for hyperproperties. In: 2020 IEEE 33rd CSF, pp. 366–379. IEEE (2020)
8. Bonakdarpour, B., et al.: Model checking timed hyperproperties in discrete-time systems. In: NASA Formal Methods Symposium, pp. 311–328. Springer (2020)
9. Bonnah, E., et al.: Model checking time window temporal logic for hyperproperties. In: 21st ACM-IEEE MEMOCODE, pp. 100–110 (2023)
10. Clarke, E., et al.: Bounded model checking using satisfiability solving. *FM* **19**(1), 7–34 (2001)
11. Clarke, E.M.: Model checking. In: FSTTCS, pp. 54–56. Springer (1997)
12. Clarkson, M.R., et al.: Temporal logics for hyperproperties. In: POST, pp. 265–284. Springer (2014)
13. Clarkson, M.R., Schneider, F.B.: Hyperproperties. *J. Comput. Secur.* **18**(6), 1157–1210 (2010)
14. Coenen, N., et al.: Verifying hyperliveness. In: CAV, pp. 121–139. Springer (2019)
15. Finkbeiner, B., et al.: Algorithms for model checking hyperltl and hyperctl. In: CAV, pp. 30–48. Springer (2015)
16. Finkbeiner, B., et al.: Specifying and verifying secrecy in workflows with arbitrarily many agents. In: ATVA, pp. 157–173. Springer (2016)
17. Finkbeiner, B., et al.: Model checking quantitative hyperproperties. In: CAV, pp. 144–163. Springer (2018)
18. Finkbeiner, B.E.A.: Eahyper: satisfiability, implication, and equivalence checking of hyperproperties. In: CAV, pp. 564–570. Springer (2017)
19. Garey, M.R., et al.: Computers and Intractability. A Guide to the (1979)
20. Goguen, J.A., et al.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, pp. 11–11. IEEE (1982)

21. Ho, H.M., et al.: On verifying timed hyperproperties. arXiv preprint [arXiv:1812.10005](https://arxiv.org/abs/1812.10005) (2018)
22. Hsu, T.H., et al.: Hyperqube: a QBF-based bounded model checker for hyperproperties. arXiv preprint [arXiv:2109.12989](https://arxiv.org/abs/2109.12989) (2021)
23. Hsu, T.H., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: TACAS, pp. 94–112. Springer (2021)
24. Knightscope. Knightscope Deploys New Autonomous Security Robot in Southern California (2022). <https://www.businesswire.com/news/home/20220316005436/en/Knightscope-Deploys-New-Autonomous-Security-Robot-in-Southern-California>
25. Nguyen, L.V., et al.: Hyperproperties of real-valued signals. In: MEMOCODE, pp. 104–113 (2017)
26. Paviot-Adet, E., et al.: Structural reductions and stutter sensitive properties. arXiv preprint [arXiv:2212.04218](https://arxiv.org/abs/2212.04218) (2022)
27. Pommellet, A., et al.: Model-checking hyperltl for pushdown systems. In: SPIN, pp. 133–152. Springer (2018)
28. Romano, S.A.: Persistent surveillance gives squadron its global purpose. <https://www.af.mil/News/Article-Display/Article/1152329/persistent-surveillance-gives-squadron-its-global-purpose/>
29. Rungta, N.: A billion SMT queries a day. In: CAV, pp. 3–18. Springer (2022)
30. Vatile, C.I., et al.: Time window temporal logic. *Theoret. Comput. Sci.* **691**, 27–54 (2017)
31. Wang, Y., et al.: Hyperproperties for robotics: planning via hyperltl. In: 2020 IEEE ICRA, pp. 8462–8468. IEEE (2020)
32. Zdancewic, S., et al.: Observational determinism for concurrent program security. In: 16th IEEE CSF, pp. 29–43. IEEE (2003)



A Tableau-Based Approach to Model Checking Linear Temporal Properties

Canh Minh Do^(✉), Tsubasa Takagi, and Kazuhiro Ogata

Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan
{canhdo,tsubasa,ogata}@jaist.ac.jp

Abstract. This paper proposes a tableau-based approach to model checking linear temporal properties to mitigate the state space explosion in model checking. The core idea of the approach is to split an original model checking problem into multiple smaller model checking problems using the tableau method and tackle each smaller one. We prove a theorem to guarantee that the multiple smaller model checking problems are equivalent to the original model checking problem. We use Maude, a high-level specification and programming language based on rewriting logic, to develop a tool called *DCA2MC* to support our approach. Some case studies are conducted to compare *DCA2MC* with Maude LTL model checker, Spin, and LTSmin model checkers in terms of running performance and memory usage, showing the power of our proposed approach.

Keywords: semantic tableaux · Linear Temporal Logic (LTL) · model checking · state space explosion · Maude

1 Introduction

Model checking [8] has become one of the most notable achievements in computer science over the past few decades. As a result, Edmund M. Clarke, together with E. Allen Emerson and Joseph Sifakis, received the 2007 ACM A. M. Turing Award, the highest distinction in the field of computer science and often regarded as the Nobel Prize of computing, for their contributions to advancing model checking into a highly effective verification technology. Model checking has been extensively used as an automatic formal verification technique in the hardware and software industries. Despite its success, model checking still has some challenges, particularly the state space explosion problem, the most annoying one. Several techniques, such as abstraction [6] and partial order reduction [7], have been devised to address this issue to some extent. Despite these efforts, existing techniques are still insufficient, making it an ongoing area of research.

Our research group proposed a divide and conquer approach to model checking some specific properties such as leads-to properties [19] expressed as $\varphi_1 \rightsquigarrow \varphi_2$,

This research was partially supported by JSPS KAKENHI Grant Numbers JP23K28060, JP23K19959, JP24K20757.

eventual properties [2] expressed as $\diamond\varphi_1$, conditional stable properties [18] expressed as $\varphi_1 \rightsquigarrow \square\varphi_2$, and until and until stable properties [10] expressed as $\varphi_1 \mathcal{U} \varphi_2$ and $\varphi_1 \mathcal{U} \square\varphi_2$, respectively, so as to deal with the state space explosion in model checking, where φ_1 and φ_2 are restricted to atomic propositions. Although these properties can be formalized in Linear Temporal Logic (LTL), we need to handle each property separately by proving a theorem to ensure the correctness of the approach for each property and developing their support tools [3, 11–13, 17, 20]. This is because it is challenging to find a single technique to deal with these different properties at once during that time. To address this challenge, we propose a tableau-based approach to model checking any linear temporal properties with no restrictions in this paper.

The tableau method is often used for checking the satisfiability and validity [23], and the automaton construction [14] of LTL formulas. In this paper, we adopt the tableau method to split the original reachable state space of a system under model checking into multiple sub-state spaces and tackle each sub-state space independently. If the size of each sub-state space is significantly smaller than the size of the original reachable state space, it would be feasible to conduct model checking experiment for the sub-state space even though it is impossible to directly conduct the model checking experiment for the original reachable state space due to the state space explosion. Thereby, our approach can mitigate the state space explosion to some extent. Moreover, our approach can use any existing LTL model checking algorithms and LTL model checkers as components because we do not alter them. We prove a theorem to guarantee that model checking problems for sub-state spaces are equivalent to the original model checking problem for the original one. An algorithm is then constructed based on the theorem so as to develop a support tool. We use Maude, a high-level specification and programming language based on rewriting logic [15], to develop a tool called `DCA2MC`¹ to support our approach. Maude [9] has the necessary facilities to develop the tool, such as LTL model checking and meta-programming, allowing us to use Maude LTL model checker as a handy software component in our implementation. We use two mutual exclusion protocols as case studies and conduct experiments to compare `DCA2MC` with Maude LTL model checker, Spin, and LTSmin model checkers in terms of running performance and memory usage. Our experimental results demonstrate the power of our approach in mitigating the state space explosion in model checking. `DCA2MC` and case studies are publicly available at <https://github.com/canhminhdo/dca2mc>.

The rest of the paper is organized as follows. Section 2 describes some preliminaries for Kripke structures. Section 3 presents the semantic tableaux to construct a tableau for an LTL formula. Section 4 illustrates multiple layer division of LTL model checking with a theorem. Section 5 constructs an algorithm based on the theorem. Section 6 presents a support tool for our approach. Section 7 shows

¹ `DCA2MC` means a divide and conquer approach to model checking. We chose this name because our tableau-based approach originates from the idea of the divide and conquer approach to model checking linear temporal properties.

some experimental results. Section 8 mentions some existing work. Section 9 finally concludes the paper with some future directions.

2 Preliminaries

We use the symbol \triangleq as “be defined as.”

Definition 1 (Kripke structures). *A Kripke structure \mathbf{K} is $\langle \mathbf{S}, \mathbf{I}, \mathbf{T}, \mathbf{A}, \mathbf{L} \rangle$ that consists of a set \mathbf{S} of states, a set $\mathbf{I} \subseteq \mathbf{S}$ of initial states, a left-total binary relation $\mathbf{T} \subseteq \mathbf{S} \times \mathbf{S}$ over states, a set \mathbf{A} of atomic propositions that contains at least one special element \top , and a labeling function $\mathbf{L} : \mathbf{S} \rightarrow 2^{\mathbf{A}}$ such that $\top \in \mathbf{L}(s)$ for each $s \in \mathbf{S}$. An element $(s, s') \in \mathbf{T}$ is called a (state) transition from s to s' and may be written as $s \rightarrow_{\mathbf{K}} s'$.*

Note that \mathbf{S} does not need to be finite. The set \mathbf{R} of reachable states is inductively defined as follows: $\mathbf{I} \subseteq \mathbf{R}$ and if $s \in \mathbf{R}$ and $(s, s') \in \mathbf{T}$, then $s' \in \mathbf{R}$. We suppose that \mathbf{R} is finite. The subscript \mathbf{K} in $s \rightarrow_{\mathbf{K}} s'$ may be omitted if it is clear from the context. We extend the labeling function \mathbf{L} to $\hat{\mathbf{L}}$ for each state s to include $\mathbf{L}(s)$ and negations of atomic propositions that do not hold at s as follows: $\hat{\mathbf{L}}(s) \triangleq \mathbf{L}(s) \cup \{\neg a \mid a \in \mathbf{A} \setminus \mathbf{L}(s)\}$.

An infinite sequence $s_0, s_1, \dots, s_i, s_{i+1}, \dots$ of states is called a path of \mathbf{K} if and only if for any natural number i , $(s_i, s_{i+1}) \in \mathbf{T}$. Let π be $s_0, s_1, \dots, s_i, s_{i+1}, \dots$ and some notations are defined as follows. For any natural numbers i and j , $\pi(i) \triangleq s_i$; $\pi^i \triangleq s_i, s_{i+1}, \dots$; $\pi_i \triangleq s_0, s_1, \dots, s_i, s_i, \dots$; $\pi_\infty \triangleq \pi$; $\pi^{(i,j)} \triangleq s_i, s_{i+1}, \dots, s_j, s_j, \dots$ if $i \leq j$ and $\pi^{(i,j)} \triangleq s_i, s_i, \dots$ otherwise; $\pi^{(i,\infty)} \triangleq \pi^i$; and $\pi_j^i \triangleq \pi^{(i,j)}$. Note that $\pi^{(0,j)} = \pi_j$. Note that $\pi_i(k) = \pi(k)$ if $k = 0, \dots, i$ and $\pi_i(k) = \pi(i)$ if $k > i$. Note that $\pi^{(i,j)}(k) = \pi(i+k)$ if $i \leq j$ and $k = 0, \dots, m$, where $j = i+m$, $\pi^{(i,j)}(k) = \pi(j)$ if $i \leq j$ and $k > j$ and $\pi^{(i,j)}(k) = \pi(i)$ if $i > j$ and k is a natural number. A path π is called a computation of \mathbf{K} if $\pi(0) \in \mathbf{I}$.

Definition 2 (Syntax of LTL). *The set \mathcal{L}_{LTL} of all formulas of Linear Temporal Logic (LTL) is generated by the following grammar:*

$$\mathcal{L}_{\text{LTL}} \ni \varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \mathcal{U} \varphi,$$

where $a \in \mathbf{A}$.

Definition 3 (Semantics of LTL). *For any Kripke structure \mathbf{K} , any path π of \mathbf{K} , and any LTL formulas φ , the satisfaction relation $\mathbf{K}, \pi \models \varphi$ is inductively defined as follows:*

1. $\mathbf{K}, \pi \models a$ if and only if $a \in \mathbf{L}(\pi(0))$;
2. $\mathbf{K}, \pi \models \neg\varphi_1$ if and only if $\mathbf{K}, \pi \not\models \varphi_1$;
3. $\mathbf{K}, \pi \models \varphi_1 \vee \varphi_2$ if and only if $\mathbf{K}, \pi \models \varphi_1$ or $\mathbf{K}, \pi \models \varphi_2$;
4. $\mathbf{K}, \pi \models \bigcirc\varphi_1$ if and only if $\mathbf{K}, \pi^1 \models \varphi_1$;

Table 1. Classification of $\alpha/\beta/X$ -formulas

α	α_1	α_2	β	β_1	β_2	X	X_1
$\neg\neg\varphi_1$	φ_1		$\neg(\varphi_1 \wedge \varphi_2)$	$\neg\varphi_1$	$\neg\varphi_2$	$\bigcirc\varphi_1$	φ_1
$\varphi_1 \wedge \varphi_2$	φ_1	φ_2	$\varphi_1 \vee \varphi_2$	φ_1	φ_2	$\neg\bigcirc\varphi_1$	$\neg\varphi_1$
$\neg(\varphi_1 \vee \varphi_2)$	$\neg\varphi_1$	$\neg\varphi_2$	$\varphi_1 \rightarrow \varphi_2$	$\neg\varphi_1$	φ_2		
$\neg(\varphi_1 \rightarrow \varphi_2)$	φ_1	$\neg\varphi_2$	$\neg(\varphi_1 \leftrightarrow \varphi_2)$	$\neg(\varphi_1 \rightarrow \varphi_2)$	$\neg(\varphi_2 \rightarrow \varphi_1)$		
$\varphi_1 \leftrightarrow \varphi_2$	$\varphi_1 \rightarrow \varphi_2$	$\varphi_2 \rightarrow \varphi_1$	$\diamond\varphi_1$	φ_1	$\bigcirc\diamond\varphi_1$		
$\square\varphi_1$	φ_1	$\bigcirc\square\varphi_1$	$\neg\square\varphi_1$	$\neg\varphi_1$	$\neg\bigcirc\square\varphi_1$		
$\neg\diamond\varphi_1$	$\neg\varphi_1$	$\neg\bigcirc\diamond\varphi_1$	$\varphi_1 \mathcal{U} \varphi_2$	φ_2	$\varphi_1 \wedge \bigcirc(\varphi_1 \mathcal{U} \varphi_2)$		
$\neg(\varphi_1 \mathcal{U} \varphi_2)$	$\neg\varphi_2$	$\neg\varphi_1 \vee \neg\bigcirc(\varphi_1 \mathcal{U} \varphi_2)$					

5. $\mathbf{K}, \pi \models \varphi_1 \mathcal{U} \varphi_2$ if and only if there exists a natural number i such that $\mathbf{K}, \pi^i \models \varphi_2$ and for each natural number $j < i$, $\mathbf{K}, \pi^j \models \varphi_1$.

where φ_1, φ_2 are any LTL formulas. Then we write $\mathbf{K} \models \varphi$ if and only if $\mathbf{K}, \pi \models \varphi$ for all computations π of \mathbf{K} .

As usual, we use the abbreviations for formulas: $\perp \triangleq \neg\top$, $\varphi_1 \wedge \varphi_2 \triangleq \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\varphi_1 \leftrightarrow \varphi_2 \triangleq (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$, $\diamond\varphi \triangleq \top \mathcal{U} \varphi$, $\square\varphi \triangleq \neg\diamond\neg\varphi$ and $\varphi_1 \rightsquigarrow \varphi_2 \triangleq \square(\varphi_1 \rightarrow \diamond\varphi_2)$. The symbols \bigcirc , \mathcal{U} , \diamond , \square , and \rightsquigarrow are called next, until, eventually, always, and leads-to temporal connectives, respectively.

Definition 4 (Literals). A literal is an atomic proposition or the negation of an atomic proposition. For each $a \in A$, the pair $(a, \neg a)$ of literals is called a complementary pair.

Definition 5 ($\alpha/\beta/X$ -formula). A formula $\varphi \in \mathcal{L}_{\text{LTL}}$ is called

- an α -formula if φ is $\neg\neg\varphi_1$, $\varphi_1 \wedge \varphi_2$, $\neg(\varphi_1 \vee \varphi_2)$, $\neg(\varphi_1 \rightarrow \varphi_2)$, $\varphi_1 \leftrightarrow \varphi_2$, $\square\varphi_1$, $\neg\diamond\varphi_1$, or $\neg(\varphi_1 \mathcal{U} \varphi_2)$.
- a β -formula if φ is $\neg(\varphi_1 \wedge \varphi_2)$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \rightarrow \varphi_2$, $\neg(\varphi_1 \leftrightarrow \varphi_2)$, $\diamond\varphi_1$, $\neg\square\varphi_1$, or $\varphi_1 \mathcal{U} \varphi_2$.
- a X -formula if φ is $\bigcirc\varphi_1$ or $\neg\bigcirc\varphi_1$.

3 Semantic Tableaux

This section shows the tableau rules for LTL and describes an algorithm to construct a semantic tableau for an LTL formula based on the tableau rules. The semantic tableau will be used for multiple layer division in LTL model checking in Sect. 4.

3.1 The Tableau Rules for LTL

The method of semantic tableaux is used as a decision procedure for satisfiability and validity in LTL. The construction of a semantic tableau for an LTL formula is to create a graph, where each node is labeled by a set of formulas, which are obtained by decomposing formulas according to the logical connectives and by expanding the temporal connectives in order to separate what has to be true immediately from the current state and what has to be true from the next state. The tableau rules are shown in Table 1. We classify them into three groups as follows: (i) an α -formula is conjunctive (except for the case $\neg\neg\varphi_1$) that has two subformulas α_1 and α_2 such that the α -formula holds if and only if both subformulas α_1 and α_2 hold, (ii) a β -formula is disjunctive that has two subformulas β_1 and β_2 such that the β -formula holds if and only if β_1 or β_2 holds, and (iii) a X -formula is a X -formula that has a subformula X_1 such that the X -formula holds if and only if X_1 holds for the next state. It suffices to define tableau rules for negation (\neg), disjunction (\vee), next (\bigcirc), and until (\mathcal{U}) connectives because other connectives are defined over them. However, we show tableau rules for other connectives in Table 1 for convenient use. The rules for the $\alpha/\beta/X$ -formulas are adopted from [4]. For more clarity, let us describe the α/β -formulas for $\Box\varphi$ and $\Diamond\varphi$ with a path π of \mathbf{K} as follows:

- $\mathbf{K}, \pi \models \Box\varphi$ if and only if $\mathbf{K}, \pi \models \varphi$ and $\mathbf{K}, \pi^1 \models \Box\varphi$.
- $\mathbf{K}, \pi \models \Diamond\varphi$ if and only if $\mathbf{K}, \pi \models \varphi$ or $\mathbf{K}, \pi^1 \models \Diamond\varphi$.

The well-known applications of the tableau method are for checking the satisfiability and validity [23] and the automaton construction [14] of LTL formulas. However, we adopt the tableau method differently to split the original reachable state space of a system under verification into multiple smaller sub-state spaces and tackle each smaller sub-state space independently so as to mitigate the state space explosion in LTL model checking in this paper.

3.2 A Tableau Construction of LTL Formulas

Given an LTL formula φ , the tableau method constructs a directed graph called a semantic tableau \mathcal{T} of φ . Each node l of \mathcal{T} is labeled with a set of formulas $U(l)$ and has one or two child nodes depending on how a formula labeling the node is decomposed or expanded. Initially, \mathcal{T} consists of a single node, the initial node, labeled with the singleton set $\{\varphi\}$, where φ is called the initial formula. First, we distinguish ordinary nodes from other nodes in \mathcal{T} as follows.

Definition 6. *We give the definition of a node in \mathcal{T} as follows:*

- *A closed node is a node such that its label contains only literals and there is at least one complementary pair.*
- *An open node is a node such that its label contains only literals and there are no complementary pairs.*
- *A state node is a node such that its label contains only literals and at least one X -formula.*

- An ordinary node is neither a closed node, an open node, nor a state node.

Definition 7 (Leaves). *The leaves of a node l in \mathcal{T} are defined as follows:*

- If l is a closed node, an open node, or a state node, l is the sole leaf of l .
- Otherwise, the leaves of l consist of all closed, open, and state nodes reachable from l in \mathcal{T} such that each path from l to these nodes does not contain any closed, open, and state nodes between l and the leaves.

Algorithm 1: Construction of a semantic tableau

Input : An LTL formula φ

Output: A semantic tableau \mathcal{T} of φ

Let \mathcal{S} be a set of nodes in \mathcal{T} being considered during the construction. Initially, \mathcal{T} consists of an initial node l_0 labeled with $U(l_0) = \{\varphi\}$ and $\mathcal{S} = \{l_0\}$. While \mathcal{S} is not empty, we take out one node l labeled with $U(l)$ from \mathcal{S} , and do as follows. In the sequel, when we say that we create a new node l' as a child of l , label l' with $U(l')$, and add l' to \mathcal{S} , it means that we create a new node l' and add l' to \mathcal{S} if \mathcal{T} does not contain a node labeled with $U(l')$. If it does, we just connect l to the existing node.

- If l is either a closed node or an open node. We skip the rest and move to the next iteration.
- If l is an ordinary node, we check as follows:
 - Let α be an α -formula and α_1, α_2 be the corresponding formulas according to Table 1. If $\alpha \in U(l)$, we create a new node l' as a child of l , label l' with

$$U(l') = (U(l) \setminus \{\alpha\}) \cup \{\alpha_1, \alpha_2\},$$

and add l' to \mathcal{S} . Note that in the case that α is $\neg\neg\varphi_1$, there is no α_2 . We then skip the rest and move to the next iteration.

- Let β be a β -formula and β_1, β_2 be the corresponding formulas according to Table 1. If $\beta \in U(l)$, we create two new nodes l' and l'' as children of l , label l' with

$$U(l') = (U(l) \setminus \{\beta\}) \cup \{\beta_1\},$$

and l'' with

$$U(l'') = (U(l) \setminus \{\beta\}) \cup \{\beta_2\},$$

and l' and l'' are added to \mathcal{S} . We then skip the rest and move to the next iteration.

- If l is a state node, we do as follows. Let the set of X -formulas in $U(l)$ be

$$\{\bigcirc\varphi_1, \dots, \bigcirc\varphi_i, \neg\bigcirc\varphi_{i+1}, \dots, \neg\bigcirc\varphi_n\}.$$

We create a new node l' as a child of l , label l' with

$$U(l') = \{\varphi_1, \dots, \varphi_i, \neg\varphi_{i+1}, \dots, \neg\varphi_n\},$$

and l' is added to \mathcal{S} .

The tableau construction terminates when \mathcal{S} is empty.

Algorithm 1 presents the construction of a semantic tableau \mathcal{T} for an LTL formula φ . It is apparent that the formulas labeling the nodes of \mathcal{T} are subformulas or negations of subformulas of φ or such formulas preceded by \bigcirc . It is trivial to prove by structural induction that the number of possible formulas labeling the nodes of \mathcal{T} is less than or equal to $2^{4 \times |\varphi|}$, where $|\varphi|$ is the length of φ . As a result, the number of nodes in \mathcal{T} is at most equal $2^{4 \times |\varphi|}$. Because $|\varphi|$ is finite and previously created nodes are used instead of creating new ones in \mathcal{T} , the construction of \mathcal{T} for any LTL formula φ terminates. The reader of interest can find the proof of the termination of the tableau construction in Appendix A.

4 Multiple Layer Division of LTL Model Checking

This section describes how we conduct multiple layer division of LTL model checking with the tableau method. First, we present a basic property of LTL formulas from the tableau construction in Algorithm 1.

Lemma 1. *The following holds.*

1. Let α be an α -formula and α_1, α_2 be the corresponding formulas according to Table 1. Then $\mathbf{K}, \pi \models \alpha$, if and only if $\mathbf{K}, \pi \models \alpha_1$ and $\mathbf{K}, \pi \models \alpha_2$.
2. Let β be a β -formula and β_1, β_2 be the corresponding formulas according to Table 1. Then $\mathbf{K}, \pi \models \beta$, if and only if $\mathbf{K}, \pi \models \beta_1$ or $\mathbf{K}, \pi \models \beta_2$.

Proof. It is immediate from the tableau construction. □

Lemma 2. *The following two conditions are equivalent:*

1. $\mathbf{K}, \pi \models \{\bigcirc\varphi_1, \dots, \bigcirc\varphi_i, \neg\bigcirc\varphi_{i+1}, \dots, \neg\bigcirc\varphi_n\}$.
2. $\mathbf{K}, \pi^1 \models \{\varphi_1, \dots, \varphi_i, \neg\varphi_{i+1}, \dots, \neg\varphi_n\}$.

Proof. It is immediate.

In the sequel, let \mathcal{T} be the semantic tableau of $\varphi \in \mathcal{L}_{\text{LTL}}$, l be a node in \mathcal{T} with label $U(l)$, and l_1, \dots, l_n be the leaves of l with labels $U(l_1), \dots, U(l_n)$, respectively. For the sake of brevity, we write as follows:

- $\mathbf{K}, \pi \models U(l)$ means that $\mathbf{K}, \pi \models \varphi$ for each $\varphi \in U(l)$.
- $\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i)$ means that $\mathbf{K}, \pi \models U(l_i)$ for some $i \in [1, n]$.

Lemma 3. $\mathbf{K}, \pi \models U(l)$ if and only if $\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i)$.

Proof. This lemma follows from Lemma 1. □

We then define some sets of formulas derived from label $U(l)$ of l as follows:

- $\text{literals}(U(l))$ is defined as the set $\{\varphi \mid \varphi \text{ is a literal, } \varphi \in U(l)\}$.
- $\text{next}(U(l))$ is defined as the set $\{\varphi \mid \bigcirc\varphi \in U(l)\} \cup \{\neg\varphi \mid \neg\bigcirc\varphi \in U(l)\}$.

That is, $\text{literals}(U(l))$ is the set of all literals in $U(l)$, and $\text{next}(U(l))$ is the set of formulas that is obtained from the X -formulas in $U(l)$ by removing the preceding next operator.

Lemma 4. *We can check whether $\mathbf{K}, \pi \models U(l)$ as follows:*

1. *If $\text{literals}(U(l_i)) \not\subseteq \hat{\mathbf{L}}(\pi(0))$ for all $i \in [1, n]$, then $\mathbf{K}, \pi \not\models U(l)$.*
2. *If $\text{literals}(U(l_i)) \subseteq \hat{\mathbf{L}}(\pi(0))$ and l_i is an open node for some $i \in [1, n]$, then $\mathbf{K}, \pi \models U(l)$.*
3. *If neither 1 nor 2 above holds, then there are some $i \in [1, n]$ such that $\text{literals}(U(l_i)) \subseteq \hat{\mathbf{L}}(\pi(0))$ and l_i is a state node. Hence,*

$$\mathbf{K}, \pi \models U(l) \Leftrightarrow \mathbf{K}, \pi^1 \models \bigvee \text{Next}_l$$

where $\text{Next}_l = \{\text{next}(U(l_i)) \mid l_i \text{ is a state node, } \text{literals}(U(l_i)) \subseteq \hat{\mathbf{L}}(\pi(0)), \text{ and } i \in [1, n]\}$.

Proof. This lemma follows from Lemma 2 and Lemma 3. \square

Definition 8 ($\text{TryEvalOne}_{\mathbf{K}, \pi}$). *Let $\text{TryEvalOne}_{\mathbf{K}, \pi}(U(l))$ be the set of sets of formulas defined by*

$$\text{TryEvalOne}_{\mathbf{K}, \pi}(U(l)) = \begin{cases} \{\{\perp\}\} & \text{if Lemma 4, 1's antecedent holds,} \\ \{\{\top\}\} & \text{if Lemma 4, 2's antecedent holds,} \\ \text{Next}_l & \text{if Lemma 4, 3's antecedent holds.} \end{cases}$$

Let us suppose that there are two artificially isolated nodes l_{\top} and l_{\perp} in \mathcal{T} with the labels $U(l_{\top}) = \{\top\}$ and $U(l_{\perp}) = \{\perp\}$, respectively. Then, it is immediate that $\text{TryEvalOne}_{\mathbf{K}, \pi}(\{\top\}) = \{\{\top\}\}$ and $\text{TryEvalOne}_{\mathbf{K}, \pi}(\{\perp\}) = \{\{\perp\}\}$.

In the sequel, let l_1, \dots, l_n be some nodes in \mathcal{T} with labels $U(l_1), \dots, U(l_n)$, respectively.

Lemma 5. *We can check whether $\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i)$ as follows:*

1. *If $\text{TryEvalOne}_{\mathbf{K}, \pi}(U(l_i)) = \{\{\perp\}\}$ for all $i \in [1, n]$, then $\mathbf{K}, \pi \not\models \bigvee_{i=1}^n U(l_i)$.*
2. *If $\text{TryEvalOne}_{\mathbf{K}, \pi}(U(l_i)) = \{\{\top\}\}$ for some $i \in [1, n]$, then $\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i)$.*
3. *If neither 1 nor 2 above holds,*

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^1 \models \bigvee \text{Next}_{l_1, \dots, l_n}$$

where $\text{Next}_{l_1, \dots, l_n} = \bigcup_{\substack{i \in [1, n] \\ \text{TryEvalOne}_{\mathbf{K}, \pi}(U(l_i)) \neq \{\{\perp\}\}}} \text{TryEvalOne}_{\mathbf{K}, \pi}(U(l_i))$.

Proof. This lemma follows from Lemma 4. \square

Definition 9 ($\text{TryEval}_{\mathbf{K}, \pi}$). *Let $\text{TryEval}_{\mathbf{K}, \pi}(\{U(l_1), \dots, U(l_n)\})$ be the set of sets of formulas defined by*

$$\text{TryEval}_{\mathbf{K}, \pi}(\{U(l_1), \dots, U(l_n)\}) = \begin{cases} \{\{\perp\}\} & \text{if Lemma 5, 1's antecedent holds,} \\ \{\{\top\}\} & \text{if Lemma 5, 2's antecedent holds,} \\ \text{Next}_{l_1, \dots, l_n} & \text{if Lemma 5, 3's antecedent holds.} \end{cases}$$

Note that $\text{TryEval}_{\mathbf{K},\pi}(\{\{\perp\}\}) = \{\{\perp\}\}$ and $\text{TryEval}_{\mathbf{K},\pi}(\{\{\top\}\}) = \{\{\top\}\}$.

Lemma 6. $\text{TryEval}_{\mathbf{K},\pi}(\{U(l_1), \dots, U(l_n)\}) = \{U(l'_1), \dots, U(l'_m)\}$ for some nodes l'_1, \dots, l'_m in \mathcal{T} .

Proof. If $\text{TryEval}_{\mathbf{K},\pi}(\{U(l_1), \dots, U(l_n)\})$ is $\{\{\perp\}\}$ or $\{\{\top\}\}$, it is immediate because l_\perp and l_\top are in \mathcal{T} . Otherwise, $\text{TryEval}_{\mathbf{K},\pi}(\{U(l_1), \dots, U(l_n)\}) = \text{Next}_{l_1, \dots, l_n}$. We consider every $\text{TryEvalOne}_{\mathbf{K},\pi}(U(l_i))$ in $\text{Next}_{l_1, \dots, l_n}$ for some $i \in [1, n]$ such that $\text{TryEvalOne}_{\mathbf{K},\pi}(U(l_i)) \neq \{\{\perp\}\}$ and $\text{TryEvalOne}_{\mathbf{K},\pi}(U(l_i)) \neq \{\{\top\}\}$ as well due to the antecedent of 3 in Lemma 5. Therefore, $\text{TryEvalOne}_{\mathbf{K},\pi}(U(l_i)) = \text{Next}_{l_i}$. Let us suppose that l_1^i, \dots, l_k^i be the leaves of l_i in \mathcal{T} . We consider every $\text{next}(l_j^i)$ in Next_{l_i} for some $j \in [1, k]$ such that l_j^i is a state node and $\text{literals}(U(l_j^i)) \subseteq \hat{\mathbf{L}}(\pi(0))$. Because l_j^i is a state node, $\text{next}(l_j^i)$ is the label of its sole child node by the tableau construction in Algorithm 1. Therefore, $\text{Next}_{l_1, \dots, l_n}$ is the set of labels of some nodes in \mathcal{T} . \square

Lemma 7. *The following holds.*

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^1 \models \bigvee \text{TryEval}_{\mathbf{K},\pi}(\{U(l_1), \dots, U(l_n)\}).$$

Proof. This lemma follows from Lemma 5. \square

For any nonzero natural number k , we consider two layer division of π , namely two paths $\pi^{(0,k)}$ and $\pi^{(k,\infty)}$. The path $\pi^{(0,k)}$ can be regarded as the first layer of the reachable state space, while the path $\pi^{(k,\infty)}$ is the second layer.

Definition 10 ($2\text{Layers}_{\mathbf{K},\pi}^k$). *For any nonzero natural number k ,*

$$2\text{Layers}_{\mathbf{K},\pi}^k(\{U(l_1), \dots, U(l_n)\}) = \text{TryEval}_{\mathbf{K},\pi^{k-1}}(\cdots (\text{TryEval}_{\mathbf{K},\pi^0}(\{U(l_1), \dots, U(l_n)\}) \cdots).$$

Lemma 8. *For any nonzero natural number k , the following holds.*

$$2\text{Layers}_{\mathbf{K},\pi}^k(\{U(l_1), \dots, U(l_n)\}) = \{U(l'_1), \dots, U(l'_m)\}$$

for some nodes l'_1, \dots, l'_m in \mathcal{T} .

Proof. We prove by induction on k .

Base Case $k = 1$. It follows from Lemma 6.

Induction Step $k \geq 2$. Let $k = k' + 1$ for some $k' \geq 1$. We have

$$2\text{Layers}_{\mathbf{K},\pi}^{k'}(\{U(l_1), \dots, U(l_n)\}) = \{U(l_1^{k'}), \dots, U(l_{m_{k'}}^{k'})\}$$

for some nodes $l_1^{k'}, \dots, l_{m_{k'}}^{k'}$ in \mathcal{T} from the induction hypothesis. Thus, it follows from the definition of $2\text{Layers}_{\mathbf{K},\pi}^{k'}$ and $2\text{Layers}_{\mathbf{K},\pi}^{k'+1}$ that

$$\begin{aligned} 2\text{Layers}_{\mathbf{K},\pi}^{k'+1}(\{U(l_1), \dots, U(l_n)\}) &= \text{TryEval}_{\mathbf{K},\pi^{k'}}(2\text{Layers}_{\mathbf{K},\pi}^{k'}(\{U(l_1), \dots, U(l_n)\})) \\ &= \text{TryEval}_{\mathbf{K},\pi^{k'}}(\{U(l_1^{k'}), \dots, U(l_{m_{k'}}^{k'})\}). \end{aligned}$$

Owing to Lemma 6, we have

$$2\text{Layers}_{\mathbf{K},\pi}^{k'+1}(\{U(l_1), \dots, U(l_n)\}) = \{U(l'_1), \dots, U(l'_m)\}$$

for some nodes l'_1, \dots, l'_m in \mathcal{T} . □

Theorem 1 (Two layer division of LTL model checking). *For any nonzero natural number k ,*

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^k \models \bigvee 2\text{Layers}_{\mathbf{K},\pi}^k(\{U(l_1), \dots, U(l_n)\}).$$

Proof. We prove by induction on k .

Base Case $k = 1$. It follows from Lemma 7.

Induction Step $k \geq 2$. Let $k = k' + 1$ for some $k' \geq 1$. We need to show

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^{k'+1} \models \bigvee 2\text{Layers}_{\mathbf{K},\pi}^{k'+1}(\{U(l_1), \dots, U(l_n)\}).$$

Owing to Lemma 8, we have

$$2\text{Layers}_{\mathbf{K},\pi}^{k'}(\{U(l_1), \dots, U(l_n)\}) = \{U(l'_1), \dots, U(l'_m)\}$$

for some nodes l'_1, \dots, l'_m from \mathcal{T} . Thus, it follows from the induction hypothesis that

$$\begin{aligned} \mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) &\Leftrightarrow \mathbf{K}, \pi^{k'} \models \bigvee 2\text{Layers}_{\mathbf{K},\pi}^{k'}(\{U(l_1), \dots, U(l_n)\}) \\ &\Leftrightarrow \mathbf{K}, \pi^{k'} \models \bigvee_{i=1}^m U(l'_i). \end{aligned}$$

Owing to Lemma 7, we have

$$\mathbf{K}, \pi^{k'} \models \bigvee_{i=1}^m U(l'_i) \Leftrightarrow \mathbf{K}, \pi^{k'+1} \models \bigvee \text{TryEval}_{\mathbf{K},\pi^{k'}}(\{U(l'_1), \dots, U(l'_m)\}).$$

From the definition of $2\text{Layers}_{\mathbf{K},\pi}^{k'}$ and $2\text{Layers}_{\mathbf{K},\pi}^{k'+1}$, we have

$$\begin{aligned} 2\text{Layers}_{\mathbf{K},\pi}^{k'+1}(\{U(l_1), \dots, U(l_n)\}) &= \text{TryEval}_{\mathbf{K},\pi^{k'}}(2\text{Layers}_{\mathbf{K},\pi}^{k'}(\{U(l_1), \dots, U(l_n)\})) \\ &= \text{TryEval}_{\mathbf{K},\pi^{k'}}(\{U(l'_1), \dots, U(l'_m)\}). \end{aligned}$$

Therefore,

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^{k'+1} \models \bigvee 2\text{Layers}_{\mathbf{K},\pi}^{k'+1}(\{U(l_1), \dots, U(l_n)\})$$

□

Corollary 1. Let φ be any LTL formula of \mathbf{K} and \mathcal{T} be the tableau of φ . For any nonzero natural number k ,

$$\mathbf{K}, \pi \models \varphi \Leftrightarrow \mathbf{K}, \pi^k \models \bigvee 2\text{Layers}_{\mathbf{K}, \pi}^k(\{\{\varphi\}\}).$$

Proof. This is an instance of Theorem 1 by replacing $\bigvee_{i=1}^n U(l_i)$ with $\bigvee\{\{\varphi\}\}$, where $\{\varphi\}$ is the label of the initial node from \mathcal{T} . \square

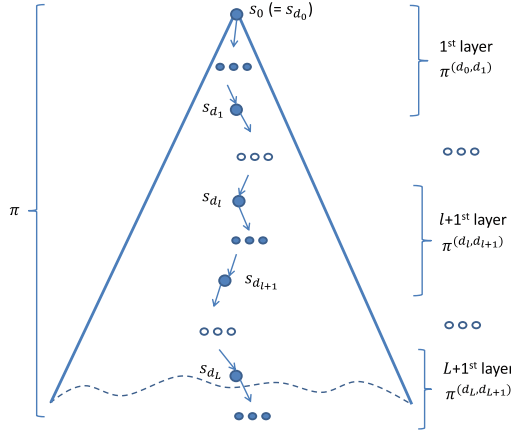


Fig. 1. $L + 1$ layer division of the reachable state space

In the sequel, let L be any nonzero natural number and $d : [0, L + 1] \rightarrow \mathbb{N}_\infty$ be a function such that $d(0) = 0$, $d(L + 1) = \infty$, and $d(l)$ is a nonzero natural number for $l \in [1, L]$, where \mathbb{N}_∞ denotes the set of natural numbers and ∞ . In addition, let d_x be $\sum_{i=0}^x d(i)$ for $i \in [0, L + 1]$. We consider $L + 1$ division of π , namely paths $\pi^{(d_0, d_1)}, \dots, \pi^{(d_l, d_{l+1})}, \dots, \pi^{(d_L, d_{L+1})}$ (see Fig. 1). The path $\pi^{(d(l), d(l+1))}$ can be regarded as the $(l + 1)$ -th layer of the reachable state space for $l \in [0, L]$. d is used to give the depth of each layer, while d_l is the depth from the top of the first layer to the bottom of the l -th layer for $l \in [0, L + 1]$.

Definition 11 ($\text{Check}_{\mathbf{K}, \pi}^L$). For any nonzero natural number L ,

$$\text{Check}_{\mathbf{K}, \pi}^L(\{U(l_1), \dots, U(l_n)\}) = \text{TryEval}_{\mathbf{K}, \pi^{d_L - 1}}(\dots(\text{TryEval}_{\mathbf{K}, \pi^0}(\{U(l_1), \dots, U(l_n)\}) \dots)).$$

Lemma 9. For any nonzero natural number L , the following holds.

$$\text{Check}_{\mathbf{K}, \pi}^L(\{U(l_1), \dots, U(l_n)\}) = \{U(l'_1), \dots, U(l'_m)\}$$

for some nodes l'_1, \dots, l'_m in \mathcal{T} .

Proof. It is similar to prove this lemma by induction on d_L as shown in Lemma 8.

Theorem 2 ($L+1$ layer division of LTL model checking). For any nonzero natural number L ,

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^{d_L} \models \bigvee \text{Check}_{\mathbf{K}, \pi}^L(\{U(l_1), \dots, U(l_n)\})$$

Proof. We prove by induction on L .

Base Case $L = 1$. It follows from Theorem 1.

Induction Step $L \geq 2$. Let $L = L' + 1$ for some $L' \geq 1$. We need to show

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^{d_{L'+1}} \models \bigvee \text{Check}_{\mathbf{K}, \pi}^{L'+1}(\{U(l_1), \dots, U(l_n)\}).$$

Let d'' be d used in $\text{Check}_{\mathbf{K}, \pi}^{L'+1}(\{U(l_1), \dots, U(l_n)\})$ such that $d''(0) = 0$, $d''(L' + 2) = \infty$, and $d''(x)$ is an arbitrary nonzero natural number for $x \in [1, L' + 1]$. The induction hypothesis is as follows:

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^{d_{L'}} \models \bigvee \text{Check}_{\mathbf{K}, \pi}^{L'}(\{U(l_1), \dots, U(l_n)\}).$$

Let d' be d used in $\text{Check}_{\mathbf{K}, \pi}^{L'}(\{U(l_1), \dots, U(l_n)\})$ such that $d'(0) = 0$, $d'(L' + 1) = \infty$, and $d'(x)$ is an arbitrary nonzero natural number for $x \in [1, L']$. Because $d''(x)$ is an arbitrary nonzero natural number for $x \in [1, L' + 1]$, we suppose that $d''(x) = d'(x)$ for $x \in [1, L']$.

Owing to Lemma 9, we have

$$\text{Check}_{\mathbf{K}, \pi}^{L'}(\{U(l_1), \dots, U(l_n)\}) = \{U(l'_1), \dots, U(l'_m)\}$$

for some nodes l'_1, \dots, l'_m from \mathcal{T} . Thus, it follows from the induction hypothesis that

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^{d_{L'}} \models \bigvee_{i=1}^m U(l'_i).$$

From the definition of $\text{Check}_{\mathbf{K}, \pi}^{L'}$ and $\text{Check}_{\mathbf{K}, \pi}^{L'+1}$, we have

$$\begin{aligned} & \text{Check}_{\mathbf{K}, \pi}^{L'+1}(\{U(l_1), \dots, U(l_n)\}) \\ &= \text{TryEval}_{\mathbf{K}, \pi^{d_{L'+1}}}(\cdots (\text{TryEval}_{\mathbf{K}, \pi^{d_{L'}}}(\text{Check}_{\mathbf{K}, \pi}^{L'}(\{U(l_1), \dots, U(l_n)\}))) \cdots) \\ &= \text{TryEval}_{\mathbf{K}, \pi^{d_{L'+1}}}(\cdots (\text{TryEval}_{\mathbf{K}, \pi^{d_{L'}}}(\{U(l'_1), \dots, U(l'_m)\})) \cdots). \end{aligned}$$

Note that $d_{L'+1} = d_{L'} + d''(L' + 1)$. Let k be an arbitrary nonzero natural number. Hence, we suppose that $k = d''(L' + 1)$. From the definition of $2\text{Layers}_{\mathbf{K}, \pi}^k$ and the result of $\text{Check}_{\mathbf{K}, \pi}^{L'+1}(\{U(l_1), \dots, U(l_n)\})$ above, we have

$$\text{Check}_{\mathbf{K}, \pi}^{L'+1}(\{U(l_1), \dots, U(l_n)\}) = 2\text{Layers}_{\mathbf{K}, \pi^{d_{L'}}}^k(\{U(l'_1), \dots, U(l'_m)\}).$$

Owing to Theorem 1, we have

$$\mathbf{K}, \pi^{d_{L'}} \models \bigvee_{i=1}^m U(l'_i) \Leftrightarrow \mathbf{K}, \pi^{d_{L'}+k} \models \bigvee \text{2Layers}_{\mathbf{K}, \pi^{d_{L'}}}^k(\{U(l'_1), \dots, U(l'_m)\}).$$

Therefore,

$$\mathbf{K}, \pi \models \bigvee_{i=1}^n U(l_i) \Leftrightarrow \mathbf{K}, \pi^{d_{L'}+1} \models \bigvee \text{Check}_{\mathbf{K}, \pi}^{L'+1}(\{U(l_1), \dots, U(l_n)\}).$$

□

Algorithm 2: The algorithm for the tableau-based approach to model checking linear temporal properties

input : \mathbf{K} – a Kripke structure
 φ – an LTL formula
 \mathcal{T} – a semantic tableau of φ constructed from Algorithm 1
 L – a nonzero natural number
 d – a function such that $d(x)$ is a nonzero natural number for $x \in [1, L]$

output: Success ($\mathbf{K} \models \varphi$) or Failure ($\mathbf{K} \not\models \varphi$)

```

1   $\mathbf{S\&F} \leftarrow \{(s, \{\{\varphi\}\}) \mid s \in \mathbf{I}\}$ 
2  forall the  $l \in [1, L]$  do
3       $\mathbf{S\&F}' \leftarrow \emptyset$ 
4      forall the  $(s, F) \in \mathbf{S\&F}$  do
5          forall the  $\pi \in P_{(\mathbf{K}, s)}^{d(l)}$  do
6               $F' \leftarrow F$ 
7              forall the  $i \in [0, d(l)]$  do
8                  if  $\text{TryEval}_{\mathbf{K}, \pi^i}(F') = \{\{\perp\}\}$  then
9                      return Failure
10                 if  $\text{TryEval}_{\mathbf{K}, \pi^i}(F') = \{\{\top\}\}$  then
11                     go to 5
12                  $F' \leftarrow \text{TryEval}_{\mathbf{K}, \pi^i}(F')$ 
13                  $\mathbf{S\&F}' \leftarrow \mathbf{S\&F}' \cup (\pi(d(l)), F')$ 
14      $\mathbf{S\&F} \leftarrow \mathbf{S\&F}'$ 
15 forall the  $(s, F) \in \mathbf{S\&F}$  do
16     forall the  $\pi \in P_{(\mathbf{K}, s)}$  do
17         if  $\mathbf{K}, \pi \not\models \bigvee F'$  then
18             return Failure
19 return Success

```

Corollary 2. Let φ be any LTL formula of \mathbf{K} and \mathcal{T} be the tableau of φ . For any nonzero natural number L ,

$$\mathbf{K}, \pi \models \varphi \Leftrightarrow \mathbf{K}, \pi^{d_L} \models \bigvee \text{Check}_{\mathbf{K}, \pi}^L(\{\{\varphi\}\}).$$

Proof. This is an instance of Theorem 2 by replacing $\bigvee_{i=1}^n U(l_i)$ with $\bigvee \{\{\varphi\}\}$, where $\{\varphi\}$ is the label of the initial node from \mathcal{T} . □

5 A Tableau-Based Approach to LTL Model Checking Algorithm

We construct Algorithm 2 based on Theorem 2 to check whether $\mathbf{K} \models \varphi$. Let $\mathbf{S\&F}$ be the set of pairs, each of which consists of a state s and a set F of sets of formulas. This means that each path that starts from the state s needs to satisfy at least one set of formulas from the set F . Recall that each set of formulas labels a node in the tableau \mathcal{T} of φ . Initially, $\mathbf{S\&F}$ is initialized with each state in \mathbf{I} and the singleton set $\{\{\varphi\}\}$. In the code fragment at lines 2–14, for each intermediate layer $l \in [1, L]$, $\mathbf{S\&F}$ is given as input for all states located at the beginning of layer l and their formulas being considered, and $\mathbf{S\&F}'$ is used to collect all states located at the bottom of layer l and their formulas. For each pair (s, F) in $\mathbf{S\&F}$, we check for each path π that starts from s up to depth $d(l)$ as follows. F' initially is assigned to F at the beginning of the iteration. We iteratively check whether $\mathbf{K}, \pi \models \bigvee F'$ can be judged at $\pi(i)$ for $i \in [0, d(l))$ with the $\text{TryEval}_{\mathbf{K}, \pi^i}$ function. If the function returns $\{\{\perp\}\}$, it is a true counterexample and **Failure** is returned immediately. If the function returns $\{\{\top\}\}$, then $\mathbf{K}, \pi \models \bigvee F'$ and so we move to check another path. Otherwise, F' is assigned to the set of sets of formulas that should be considered for π^{i+1} at line 12. At the end of the iteration, if $\mathbf{K}, \pi \models \bigvee F'$ cannot be judged at any $\pi(i)$ for $i \in [0, d(l))$, we add the pair $(\pi(d(l)), F')$ to $\mathbf{S\&F}'$ to be considered for the next layer. If no **Failure** is returned during checking for layer l , $\mathbf{S\&F}'$ will be assigned to $\mathbf{S\&F}$ as the input for the next layer at line 14. The code fragment at lines 15–18 checks for each pair (s, F) in $\mathbf{S\&F}$ and for each path π that starts from s with F at the final layer, saying $(L + 1)$ -th layer. If $\mathbf{K}, \pi \not\models \bigvee F$, **Failure** is returned. Otherwise, **Success** is returned at the end.

6 Implementation

We implemented the tableau-based approach to model checking linear temporal properties based on Algorithm 2 in Maude as **DCA2MC**, which is publicly available at <https://github.com/canhminhdo/dca2mc>. Maude [9] is a high-level specification and programming language based on rewriting logic [15]. It has the necessary facilities to develop **DCA2MC**, such as LTL model checking and meta-programming, allowing us to use Maude LTL model checker as a handy software component in our implementation. Therefore, we use Maude as a formal specification, its model checker, and meta-programming for our tool development.

DCA2MC provides an interactive mode for users to conduct model checking experiments with our approach using the following commands, where underscores denote arguments in the commands.

- `initialize[_ , _ , _]` to initialize the application with a system module identifier, an initial state, and an LTL formula under model checking given as inputs. A tableau for the LTL formula is constructed at this initialization.

Table 2. Experimental results for the leads-to property $\text{inWs1} \rightsquigarrow \text{inCs1}$

Protocol	Layers	DCA2MC	Maude LTL model checker	Spin	LTSmin
Qlock (10 processes)	2 2	5 h 4 m	24 d 20 h 11 m	OOM (after 10 h 9 m)	OOM (after 5 h 9 m)
Anderson (9 processes)	2 2	59 m 7 s	11 d 19 h 17 m	OOM (after 3 h 54 m)	OOM (after 6 h 45 m)

- `layerCheck _` to generate states located at the beginning of the final layer together with their formulas for a given layer configuration, which is a list of nonzero natural numbers to denote the depth of each intermediate layer. This generation is performed using the tableau method as shown in Algorithm 2 for intermediate layers.
- `lastCheck` to conduct model checking experiments for each state and its formulas at the final layer as shown in Algorithm 2 for the final layer. It is important to note that the model checking experiments at the final layer are carried out by Maude LTL model checker in our implementation.
- `quit` to terminate the application.

DCA2MC returns either `success` or `failure` to indicate whether the system under model checking satisfies the LTL formula starting from an initial state.

7 Experiments

7.1 Experiment Setup

We used a MacPro computer that carries a 2.5 GHz microprocessor with 28 cores and 768 GB memory of RAM to conduct experiments. We use two mutual exclusion protocols as systems under model checking: Qlock with 10 process participants and Anderson with 9 process participants. Qlock is an abstract version of the Dijkstra binary semaphore, while Anderson is an array-based mutual exclusion protocol invented by Anderson [1]. We suppose that each process enters the critical section once and then moves to the final section and stays there forever to avoid long lasso loops in our specifications. Let `inWs1`, `inCs1`, and `inFs1` be atomic propositions that hold at a state if and only if a certain process resides in the waiting, critical, and final sections, respectively. We consider two commonly used liveness properties for our case studies: a leads-to property expressed as $\text{inWs1} \rightsquigarrow \text{inCs1}$ and an eventual property expressed as $\diamond \text{inFs1}$. The former property states that whenever a process resides in the waiting section, it will eventually enter the critical section, while the latter property states that a process eventually resides in the final section. We then conduct experiments for Qlock and Anderson with the leads-to and eventual properties to compare DCA2MC with Maude LTL model checker (version 3.2), Spin (version 6.5.1), and LTSmin (version 3.0.2) in terms of running performance and memory

usage. Note that we prepare the Maude specifications of Qlock and Anderson for Maude LTL model checker and DCA2MC that are available at <https://github.com/canhminhdo/dca2mc>. Meanwhile, we prepare the corresponding Promela specifications of Qlock and Anderson for Spin and these Promela specifications can also be fed into LTSmin using a so-called translator SpinS [21]. The Promela specifications are publicly available at <https://github.com/canhminhdo/promela-specs>.

Table 3. Experimental results for the eventual property $\diamond \text{inFs1}$

Protocol	Layers	DCA2MC	Maude LTL model checker	Spin	LTSmin
Qlock (10 processes)	2 2	2 h 51 m	14 d 4 h 41 m	OOM (after 9 h 18 m)	OOM (after 5 h 12 m)
Anderson (9 processes)	2 2	21 m 27 s	1 d 10 h 37 m	50 s	6 m 18 s

7.2 Experimental Results

The experimental results for model checking Qlock and Anderson with the leads-to property $\text{inWs1} \rightsquigarrow \text{inCs1}$ and the eventual property $\diamond \text{inFs1}$ are shown in Table 2 and Table 3, respectively. The second column shows the layer configuration used only for DCA2MC. Other model checkers do not use this information. The OOM (Out of Memory) value in the fifth and sixth columns denotes that 768 GB of memory was insufficient for Spin and LTSmin to conduct model checking experiments due to the state space explosion.

DCA2MC vs. Maude LTL Model Checker . For Qlock with 10 processes and Anderson with 9 processes, DCA2MC could complete the model checking experiments significantly faster than Maude LTL model checker for the two properties. It is important to note that DCA2MC is implemented in Maude and uses Maude LTL model checker as a software component to conduct model checking experiments at the final layer. Moreover, some extra costs are introduced to DCA2MC due to its performance at the meta-level, while Maude LTL model checker is performed at the object level. Therefore, these results demonstrate the power of our approach and show that dealing with many smaller sub-state spaces can be more efficient than dealing with the original state space because it does not need to manage a huge amount of memory, the size of a hash table is also smaller and so state matching and storing are less burdensome, and it is also good for hardware cache.

DCA2MC vs. Spin and LTSmin . For Qlock with 10 processes, Spin and LTSmin model checkers ran out of memory after several hours for the two properties. For Anderson with 9 processes, Spin and LTSmin ran out of memory for

the leads-to property but not for the eventual property. Meanwhile, both Maude LTL model checker and DCA2MC could deal with them. Especially, DCA2MC could deal with them effectively. These results demonstrate the power of our approach in mitigating the state space explosion and improving the running performance of model checking when dealing with large state spaces. Spin and LTSmin could handle Anderson with 9 processes for the eventual property but struggled for the leads-to property. This is because the eventual property is simpler than the leads-to property, making the size of the product automaton of the system under verification and the negation of the eventual property significantly smaller, as the time complexity of LTL model checking is $O(|S| \times 2^{O(|\varphi|)})$ [22], where $|S|$ is the size of the system, and $|\varphi|$ is the length of φ , the desired property.

For Anderson with the eventual property, Spin and LTSmin could complete the model checking experiments surprisingly fast, while Maude LTL model checker and DCA2MC need to spend much more time to complete their verification. That is because Spin and LTSmin use a state vector of variables to store states efficiently in memory and are equipped with many optimization techniques to reduce the state space, especially the partial order reduction [7], while Maude does not have and neither does DCA2MC. The key strength of DCA2MC lies in its ability to split the original reachable state space into smaller sub-state spaces and tackle each sub-state space independently. This makes our approach possible to handle larger state spaces than other model checkers. These findings suggest that we should implement our approach into existing efficient model checkers, such as Spin and LTSmin, and use them as an efficient software component in our implementation. This would allow us to leverage the advantage of our approach and the optimization techniques used in efficient model checkers, making it possible to handle larger state spaces effectively. This would be one piece of our future work.

8 Related Work

SAT/SMT-based bounded model checking (SAT/SMT-BMC) [5] is a highly effective method for addressing the state space explosion issue in model checking. This technique can identify a counterexample near an initial state, but it cannot prove that a system enjoys desired properties. To overcome this limitation, k -induction [16], an extension of SAT/SMT-BMC, was devised. This method uses mathematical induction in conjunction with SAT/SMT-BMC to verify that a system enjoys its desired properties. An SAT/SMT solver checks the desired properties up to a certain bounded depth from an arbitrary initial state, which is considered the base case in the mathematical induction. For each state sequence up to the bounded depth, all successor states from the last state of the sequence are verified with the desired properties using an SAT/SMT solver, constituting the induction step. Our approach shares the fundamental concept with SAT/SMT-BMC as we perform bounded model checking experiments for sub-state spaces at each intermediate layer. Additionally, our approach can also be seen as an extension of BMC because we conduct unbounded model checking

for sub-state spaces in the final layer to verify the desired properties, without using SAT/SMT solvers. The limitation of our approach is that it cannot handle long lasso loops in specifications because these make the sub-state spaces in the final layer more likely to be the same size as the original reachable state space. Dealing with this limitation would be one piece of our future work.

9 Conclusion

We have described the tableau-based approach to model checking linear temporal properties by showing how to construct a tableau for an LTL formula, how to split an original model checking problem into smaller model checking problems using the tableau method, and how to tackle each smaller one independently. We have proved a theorem to guarantee the correctness of our approach, based on which an algorithm has been constructed to develop the support tool. We have used Maude to develop the support tool called DCA2MC and have conducted some experiments to compare DCA2MC with Maude LTL model checker, Spin, and LTSmin model checkers in terms of the running performance and memory usage. The experimental results have demonstrated the power of our approach in mitigating the state space explosion and improving the running performance of model checking. In addition to some future work mentioned earlier, we plan to conduct more case studies and develop a parallel version of DCA2MC to improve the running performance of model checking further because the model checking experiment for each sub-state space in the final layer is completely independent.

A The Termination of the Tableau Construction

To show the termination property of the tableau construction, we first define the length of formulas and the set of subformulas of formulas.

Definition 12. *The length $|\varphi|$ of $\varphi \in \mathcal{L}_{\text{LTL}}$ is defined inductively as follows:*

1. $|a| = 1$ for each $a \in \mathbf{A}$;
2. $|\neg\varphi| = |\varphi| + 1$;
3. $|\varphi_1 \vee \varphi_2| = |\varphi_1| + |\varphi_2| + 1$;
4. $|\bigcirc\varphi| = |\varphi| + 1$;
5. $|\varphi_1 \mathcal{U} \varphi_2| = |\varphi_1| + |\varphi_2| + 1$.

Definition 13. *The set $\text{Sub}(\varphi)$ of subformulas of $\varphi \in \mathcal{L}_{\text{LTL}}$ is defined inductively as follows:*

1. $\text{Sub}(a) = \{a\}$ for each $a \in \mathbf{A}$;
2. $\text{Sub}(\neg\varphi) = \{\neg\varphi\} \cup \text{Sub}(\varphi)$;
3. $\text{Sub}(\varphi_1 \vee \varphi_2) = \{\varphi_1 \vee \varphi_2\} \cup \text{Sub}(\varphi_1) \cup \text{Sub}(\varphi_2)$;
4. $\text{Sub}(\bigcirc\varphi) = \{\bigcirc\varphi\} \cup \text{Sub}(\varphi)$;
5. $\text{Sub}(\varphi_1 \mathcal{U} \varphi_2) = \{\varphi_1 \mathcal{U} \varphi_2\} \cup \text{Sub}(\varphi_1) \cup \text{Sub}(\varphi_2)$.

We define some sets of formulas as follows:

- $\text{Sub}_\neg(\varphi) \triangleq \{\neg\psi \mid \psi \in \text{Sub}(\varphi)\}$ is the negations of $\text{Sub}(\varphi)$,
- $\text{Sub}_\bigcirc(\varphi) \triangleq \{\bigcirc\psi \mid \psi \in \text{Sub}(\varphi) \cup \text{Sub}_\neg(\varphi)\}$ is the formulas of $\text{Sub}(\varphi) \cup \text{Sub}_\neg(\varphi)$ preceded by \bigcirc ,
- $\mathcal{F}(\varphi) \triangleq \text{Sub}(\varphi) \cup \text{Sub}_\neg(\varphi) \cup \text{Sub}_\bigcirc(\varphi)$.

We then prove the relation between the size of $\mathcal{F}(\varphi)$ and the length of φ .

Lemma 10. $|\mathcal{F}(\varphi)| \leq 4 \times |\varphi|$.

Proof. We prove it by structural induction on φ as follows:

Base Case $\varphi = a \in \mathbf{A}$. Because

$$\mathcal{F}(a) = \{a, \neg a, \bigcirc a, \bigcirc \neg a\} \text{ and } |a| = 1,$$

we have $|\mathcal{F}(a)| = 4 \leq 4 \times |a|$.

Induction Step

Case II $\varphi = \neg\varphi_1$. We observe that

$$\begin{aligned} \text{Sub}_\neg(\neg\varphi_1) &= \{\neg\psi \mid \psi \in \text{Sub}(\neg\varphi_1)\} \\ &= \{\neg\psi \mid \psi \in \text{Sub}(\varphi_1)\} \cup \{\neg\neg\varphi_1\} \\ &= \text{Sub}_\neg(\varphi_1) \cup \{\neg\neg\varphi_1\}, \\ \text{Sub}_\bigcirc(\neg\varphi_1) &= \{\bigcirc\psi \mid \psi \in \text{Sub}(\neg\varphi_1) \cup \text{Sub}_\neg(\neg\varphi_1)\} \\ &= \{\bigcirc\psi \mid \psi \in \text{Sub}(\varphi_1) \cup \text{Sub}_\neg(\varphi_1)\} \cup \{\bigcirc\neg\varphi_1\} \cup \{\bigcirc\neg\neg\varphi_1\} \\ &= \text{Sub}_\bigcirc(\varphi_1) \cup \{\bigcirc\neg\varphi_1\} \cup \{\bigcirc\neg\neg\varphi_1\}. \end{aligned}$$

Thus,

$$\begin{aligned} \mathcal{F}(\neg\varphi_1) &= \text{Sub}(\neg\varphi_1) \cup \text{Sub}_\neg(\neg\varphi_1) \cup \text{Sub}_\bigcirc(\neg\varphi_1) \\ &= \{\neg\varphi_1\} \cup \text{Sub}(\varphi_1) \cup \text{Sub}_\neg(\varphi_1) \\ &\quad \cup \{\neg\neg\varphi_1\} \cup \text{Sub}_\bigcirc(\varphi_1) \cup \{\bigcirc\neg\varphi_1\} \cup \{\bigcirc\neg\neg\varphi_1\} \\ &= \mathcal{F}(\varphi_1) \cup \{\neg\varphi_1, \neg\neg\varphi_1, \bigcirc\neg\varphi_1, \bigcirc\neg\neg\varphi_1\}. \end{aligned}$$

By the induction hypothesis, $|\mathcal{F}(\varphi_1)| \leq 4 \times |\varphi_1|$. Hence,

$$|\mathcal{F}(\neg\varphi_1)| = |\mathcal{F}(\varphi_1)| + 4 \leq 4 \times |\varphi_1| + 4 = 4 \times (|\varphi_1| + 1) = 4 \times |\neg\varphi_1|.$$

Case I2 $\varphi = \varphi_1 \vee \varphi_2$. We observe that

$$\begin{aligned}
\text{Sub}_{\neg}(\varphi_1 \vee \varphi_2) &= \{\neg\psi \mid \psi \in \text{Sub}(\varphi_1 \vee \varphi_2)\} \\
&= \{\neg\psi \mid \psi \in \text{Sub}(\varphi_1) \cup \text{Sub}(\varphi_2)\} \cup \{\neg(\varphi_1 \vee \varphi_2)\} \\
&= \{\neg\psi \mid \psi \in \text{Sub}(\varphi_1)\} \cup \{\neg\psi \mid \psi \in \text{Sub}(\varphi_2)\} \cup \{\neg(\varphi_1 \vee \varphi_2)\} \\
&= \text{Sub}_{\neg}(\varphi_1) \cup \text{Sub}_{\neg}(\varphi_2) \cup \{\neg(\varphi_1 \vee \varphi_2)\}, \\
\text{Sub}_{\circ}(\varphi_1 \vee \varphi_2) &= \{\circ\psi \mid \psi \in \text{Sub}(\varphi_1 \vee \varphi_2) \cup \text{Sub}_{\neg}(\varphi_1 \vee \varphi_2)\} \\
&= \{\circ\psi \mid \psi \in \text{Sub}(\varphi_1) \cup \text{Sub}(\varphi_2) \cup \text{Sub}_{\neg}(\varphi_1) \cup \text{Sub}_{\neg}(\varphi_2)\} \\
&\quad \cup \{\circ(\varphi_1 \vee \varphi_2)\} \cup \{\circ\neg(\varphi_1 \vee \varphi_2)\} \\
&= \{\circ\psi \mid \psi \in \text{Sub}(\varphi_1) \cup \text{Sub}_{\neg}(\varphi_1)\} \\
&\quad \cup \{\circ\psi \mid \psi \in \text{Sub}(\varphi_2) \cup \text{Sub}_{\neg}(\varphi_2)\} \\
&\quad \cup \{\circ(\varphi_1 \vee \varphi_2)\} \cup \{\circ\neg(\varphi_1 \vee \varphi_2)\} \\
&= \text{Sub}_{\circ}(\varphi_1) \cup \text{Sub}_{\circ}(\varphi_2) \cup \{\circ(\varphi_1 \vee \varphi_2)\} \cup \{\circ\neg(\varphi_1 \vee \varphi_2)\}.
\end{aligned}$$

Thus,

$$\begin{aligned}
\mathcal{F}(\varphi_1 \vee \varphi_2) &= \text{Sub}(\varphi_1 \vee \varphi_2) \cup \text{Sub}_{\neg}(\varphi_1 \vee \varphi_2) \cup \text{Sub}_{\circ}(\varphi_1 \vee \varphi_2) \\
&= \{\varphi_1 \vee \varphi_2\} \cup \text{Sub}(\varphi_1) \cup \text{Sub}(\varphi_2) \\
&\quad \cup \text{Sub}_{\neg}(\varphi_1) \cup \text{Sub}_{\neg}(\varphi_2) \cup \{\neg(\varphi_1 \vee \varphi_2)\} \\
&\quad \cup \text{Sub}_{\circ}(\varphi_1) \cup \text{Sub}_{\circ}(\varphi_2) \cup \{\circ(\varphi_1 \vee \varphi_2)\} \cup \{\circ\neg(\varphi_1 \vee \varphi_2)\} \\
&= \mathcal{F}(\varphi_1) \cup \mathcal{F}(\varphi_2) \cup \{\varphi_1 \vee \varphi_2, \neg(\varphi_1 \vee \varphi_2), \circ(\varphi_1 \vee \varphi_2), \circ\neg(\varphi_1 \vee \varphi_2)\}.
\end{aligned}$$

By the induction hypothesis, $|\mathcal{F}(\varphi_1)| \leq 4 \times |\varphi_1|$ and $|\mathcal{F}(\varphi_2)| \leq 4 \times |\varphi_2|$. Hence,

$$|\mathcal{F}(\varphi_1 \vee \varphi_2)| = |\mathcal{F}(\varphi_1)| + |\mathcal{F}(\varphi_2)| + 4 \leq 4 \times (|\varphi_1| + |\varphi_2| + 1) = 4 \times |\varphi_1 \vee \varphi_2|.$$

Case I3 $\varphi = \circ\varphi_1$. The proof is similar to **Case I1**.

Case I4 $\varphi = \varphi_1 \mathcal{U} \varphi_2$. The proof is similar to **Case I2**. \square

Let \mathcal{T} be the tableau of φ constructed based on Algorithm 1 with the tableau rules. It is apparent that the formulas labeling the nodes of \mathcal{T} are subformulas or negations of subformulas of φ or such formulas preceded by \circ , that is $\mathcal{F}(\varphi)$. Therefore, the number of nodes of \mathcal{T} is at most equal to the number of subsets of $\mathcal{F}(\varphi)$, that is $2^{|\mathcal{F}(\varphi)|} \leq 2^{4 \times |\varphi|}$ regarding Lemma 10. Because $|\varphi|$ is finite and previously created nodes are used instead of creating new ones in \mathcal{T} , the construction of \mathcal{T} for any LTL formula φ terminates.

References

1. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* **1**(1), 6–16 (1990)
2. Aung, M.N., Phyo, Y., Do, C.M., Ogata, K.: A divide and conquer approach to eventual model checking. *Mathematics* **9**(4) (2021)
3. Aung, M.N., Phyo, Y., Do, C.M., Ogata, K.: A tool for model checking eventual model checking in a stratified way. In: 9th DSA, pp. 270–279 (2022)
4. Ben-Ari, M.: *Mathematical Logic for Computer Science*. Springer, Heidelberg (2012). <https://doi.org/10.1007/978-1-4471-4129-7>
5. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. *Formal Methods Syst. Des.* **19**(1), 7–34 (2001)
6. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* **16**(5), 1512–1542 (1994)
7. Clarke, E.M., Grumberg, O., Minea, M., Peled, D.A.: State space reduction using partial order techniques. *Int. J. Softw. Tools Technol. Transf.* **2**(3), 279–287 (1999)
8. Marques-Silva, J., Malik, S.: Propositional SAT solving. In: *Handbook of Model Checking*, pp. 247–275. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_9
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: *All About Maude - A High-Performance Logical Framework*. LNCS, vol. 4350. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71999-1>
10. Do, C.M., Phyo, Y., Ogata, K.: A divide & conquer approach to until and until stable model checking. In: 34th SEKE (2022)
11. Do, C.M., Phyo, Y., Ogata, K.: Sequential and parallel tools for model checking conditional stable properties in a layered way. *IEEE Access* **10**, 133749–133765 (2022)
12. Do, C.M., Phyo, Y., Riesco, A., Ogata, K.: Optimization techniques for model checking leads-to properties in a stratified way. *ACM Trans. Softw. Eng. Methodol.* **32**(6) (2023)
13. Do, C.M., Phyo, Y., Riesco, A., Ogata, K.: A parallel stratified model checking technique/tool for leads-to properties. In: 7th ISSSR, pp. 155–166 (2021)
14. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV 1995. IAICT, pp. 3–18. Springer, Boston, MA (1996). https://doi.org/10.1007/978-0-387-34892-6_1
15. Meseguer, J.: Twenty years of rewriting logic. *J. Log. Algebraic Methods Program.* **81**(7–8), 721–781 (2012)
16. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: from refutation to verification. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_2
17. Phyo, Y., Aung, M.N., Do, C.M., Ogata, K.: A layered and parallelized method of eventual model checking. *Information* **14**(7), 384 (2023)
18. Phyo, Y., Do, C.M., Ogata, K.: A divide & conquer approach to conditional stable model checking. In: Cerone, A., Ölveczky, P.C. (eds.) ICTAC 2021. LNCS, vol. 12819, pp. 105–111. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85315-0_7
19. Phyo, Y., Do, C.M., Ogata, K.: A divide & conquer approach to leads-to model checking. *Comput. J.* **65**, 1353–1364 (2021)

20. Phyto, Y., Do, C.M., Ogata, K.: A support tool for the L+1-layer divide & conquer approach to leads-to model checking. In: COMPSAC, pp. 854–863. IEEE (2021)
21. van der Berg, F., Laarman, A.: SpinS: extending LTSmin with Promela through SpinJa. *Electron. Notes Theor. Comput. Sci.* **296**, 95–105 (2013)
22. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Moller, F., Birtwistle, G. (eds.) *Logics for Concurrency*. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60915-6_6
23. Wolper, P.: The tableau method for temporal logic: an overview. *Logique et Anal. (N.S.)* **28**(110/111), 119–136 (1985)



Simple LTL Model Checking on Finite and Infinite Traces over Concrete Domains

David Doose^(✉)  and Julien Brunel 

DTIS, ONERA, Université de Toulouse, Toulouse, France
{david.doose,julien.brunel}@onera.fr

Abstract. There exist different semantics for Linear Temporal Logic (LTL) in terms of finiteness of the considered traces. Although several ones can be useful depending on the verification context, no verification framework handle their diversity in a simple way. Another limitation of current LTL verification tools is the treatment of concrete domains (bounded and infinite integers, real numbers, etc.). We present an approach to LTL model checking on both finite and infinite traces with concrete domains. Our method is based on an SMT solver and on Bounded Model Checking (BMC). We also present some experiments and compare our tool with NuSMV and nuXmv.

Keywords: Bounded Model Checking · LTL · Finite traces · concrete domains

1 Introduction

Temporal logic model checking has proven useful for verifying behavioral properties in software and hardware systems. Over the last decades, many works have proposed new algorithms, especially using SAT encoding. These advances in SAT and SMT solvers have led to model checkers capable of handling industrial-size systems.

A first limitation is that most techniques assume that temporal formulas are built from atomic propositions. However, in many case studies, using concrete domains such as integers, floats, and real numbers would be very helpful.

A second limitation is the restricted temporal semantics. For Linear Temporal Logic (LTL), the usual semantics only considers infinite traces. However, it is useful to reason about finite and truncated executions. Even in reactive systems, some executions may end due to crashes or expected behavior termination, and truncated executions can help simulate the system or find simple counter-examples. Although finite semantics are defined in the literature [1, 2], there is no unified framework that allows users to easily combine these different types of reasoning.

We claim that, given the current state of the art in SMT solvers and Bounded Model Checking (BMC) algorithms, it is possible to handle these two limitations

to a reasonable extent. We also found that a simple criterion for deciding whether a completeness threshold (in terms of trace length) has been reached during BMC can be computed by SMT solvers in reasonable time, providing a complete model checking algorithm. The main contribution of this article is to propose a framework for performing LTL model checking that is:

- usable over concrete domains,
- considering three possible trace semantics (infinite, truncated, and maximal finite),
- with a completeness option (on the length of traces) that also guarantees termination under the assumption of finite domains.

We implemented a prototype and evaluated its tractability with different simple, scalable examples. By comparing our prototype with the NuSMV and nuXmv implementations of the BMC and IC3 algorithms, we observed good performance, especially for models using integer numbers.

In Sect. 2, we discuss the different possible trace semantics. Section 3 presents the main principles of our solver. Section 4 shows some experiments and compare our tool with NuSMV and nuXmv.

2 Trace Semantics

In this section, we develop on different semantics for LTL, which can all be of interest depending on the verification purpose. Each one relies on a specific kind of traces: infinite traces, truncated traces and maximal finite traces.

Remark 1. In this section, we address temporal semantics and do not deal with concrete domains. This is why we consider an abstract set of states and atomic propositions, which are associated with states by the valuation V . In the next sections, in which we will consider concrete domains, a state will be implicitly defined by a mapping from variables to values (in the concrete domain).

2.1 Traces

The verification problem that we address assumes that the system under study is modeled as a transition system. Thus, the traces that are considered in our approach come from this transition system.

Definition 1 (Transition system, traces). *Given a set P of atomic propositions, a transition system (TS) over P is defined as a tuple (S, I, T, V) where*

- S is a (possibly infinite) set of states,
- $I \subseteq S$ is the set of initial states,
- $T \subseteq S \times S$ is the transition relation,
- $V : S \rightarrow 2^P$ is a valuation function associating each state with the set of atomic propositions that are true in this state.

Given such a TS ,

- an infinite trace is defined as an infinite sequence of states s_0, s_1, \dots such that $s_0 \in I$ and $\forall i \in \mathbb{N} \quad (s_i, s_{i+1}) \in T$,
- a truncated trace is a finite sequence of states s_0, s_1, \dots, s_n such that $s_0 \in I$ and $\forall i \in 0..n-1 \quad (s_i, s_{i+1}) \in T$,
- a maximal finite trace is a truncated trace s_0, s_1, \dots, s_n such that $\forall s \in S \quad (s_n, s) \notin T$.

2.2 LTL Syntax and Semantics

Definition 2 (LTL syntax). Given a set P of atomic propositions, LTL formulas are defined inductively by the following grammar:

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U}\varphi$$

where \top denotes the constant true and $p \in P$ is an atomic proposition.

\mathbf{X} and \mathbf{U} stand for the “next” and “until” connectives. All the usual Boolean connectives can be defined in terms of the negation (\neg) and the conjunction (\wedge) in the natural way. The standard temporal connectives \mathbf{F} (eventually) and \mathbf{G} (always) can be defined in terms of \mathbf{U} as $\mathbf{F}\varphi = \top \mathbf{U}\varphi$ and $\mathbf{G}\varphi = \neg \mathbf{F}\neg\varphi$.

The standard semantics of LTL, which is presented for instance in [3, 4], only considers infinite traces. This is also the semantics that is considered in most of the verification tools handling temporal logic such as the BDD-based algorithm in NuSMV and nuXmv [5, 6], Alloy Analyzer [7, 8], and TLA/TLC [9].

Definition 3 (LTL semantics on infinite traces). Given an infinite trace $\sigma = s_0, s_1, \dots$ and a valuation function $V : S \rightarrow 2^P$, the satisfaction relation \models_{inf} is defined by induction on formulas, for any $i \in \mathbb{N}$:

- $\sigma, i \models_{inf} \top$
- $\sigma, i \models_{inf} p$ if $p \in V(s_i)$
- $\sigma, i \models_{inf} \neg\varphi$ if $\sigma, i \not\models_{inf} \varphi$
- $\sigma, i \models_{inf} \varphi_1 \wedge \varphi_2$ if $\sigma, i \models_{inf} \varphi_1$ and $\sigma, i \models_{inf} \varphi_2$
- $\sigma, i \models_{inf} \mathbf{X}\varphi$ if $\sigma, i+1 \models_{inf} \varphi$
- $\sigma, i \models_{inf} \varphi_1 \mathbf{U}\varphi_2$ if $\exists j \geq i$ such that $\sigma, j \models_{inf} \varphi_2$ and $\forall i \leq k < j \quad \sigma, k \models_{inf} \varphi_1$

In some contexts, such as simulation, reasoning about a finite execution traces can be useful. Some works have already studied LTL semantics in such a case [1, 2]. An important issue is the way we deal with the last state. The most commonly adopted semantics in this case is the following.

Definition 4 (LTL semantics on finite traces). Given a finite trace $\sigma = s_0, s_1, \dots, s_n$ and a valuation function $V : S \rightarrow 2^P$, the satisfaction relation \models_{fin} is defined by induction on formulas, for any $i \in 0..n$ (the semantics of Boolean connectives and constants is the same as for the infinite case):

- $\sigma, i \models_{fin} \mathbf{X}\varphi$ if $i < n$ and $\sigma, i + 1 \models_{fin} \varphi$
- $\sigma, i \models_{fin} \varphi_1 \mathbf{U}\varphi_2$ if $\exists j$ such that $i \leq j \leq n$ and $\sigma, j \models_{fin} \varphi_2$ and $\forall i \leq k < j, \sigma, k \models_{fin} \varphi_1$

Now that we provided a meaning to the satisfaction of an LTL formula by a trace, we can define the satisfaction of a formula by a transition system. In fact, we propose to distinguish between three possible satisfaction relations.

Definition 5 (LTL satisfaction by a transition system). *Given a transition system \mathcal{M} and an LTL formula φ , we consider the following three satisfaction relations:*

- \mathcal{M} satisfies φ on infinite traces (written $\mathcal{M} \models_{inf} \varphi$) if for every infinite trace σ of \mathcal{M} , $\sigma, 0 \models_{inf} \varphi$.
- \mathcal{M} satisfies φ on truncated traces (written $\mathcal{M} \models_{trun} \varphi$) if for every truncated trace σ of \mathcal{M} , $\sigma, 0 \models_{fin} \varphi$.
- \mathcal{M} satisfies φ on maximal traces (written $\mathcal{M} \models_{max} \varphi$) if for every infinite trace σ of \mathcal{M} , $\sigma, 0 \models_{inf} \varphi$ and for every maximal finite trace σ , $\sigma, 0 \models_{fin} \varphi$.

2.3 Illustration

In order to illustrate the different semantics for LTL, we consider the simple transition system shown in Fig. 1, including three states, three transitions and two atomic propositions x and y . Let us call \mathcal{M} this transition system.

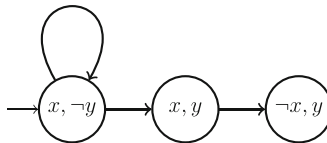


Fig. 1. Simple transition system

Infinite Trace Semantics. If we only consider infinite traces, in this example, only the first loop stands, and the model can be illustrated as shown in Fig. 2a. As a consequence, the formula $\mathbf{G}x$ will be true on all traces under an infinite semantics, i.e., $\mathcal{M} \models_{inf} \mathbf{G}x$.

Truncated Trace Semantics. The *truncated* semantics considers any path in the graph to be an acceptable trace. In particular, the trace shown in the Fig. 2b, which ends in the state satisfying $x \wedge y$, is an acceptable truncated trace. Let us call σ this trace. We have that $\sigma, 0 \models_{fin} \mathbf{F}\mathbf{G}(x \wedge y)$. Therefore, the negation of this formula is not satisfied by the transition system according to the truncated semantics: $\mathcal{M} \not\models_{trun} \neg\mathbf{F}\mathbf{G}(x \wedge y)$.

Maximal Finite Semantics. Now, if we consider traces that are maximal finite, the trace that ends in the state satisfying $\neg x \wedge y$ is taken into account (as shown by Fig. 2c) but not the truncated trace ending in the middle state, which satisfies $x \wedge y$. So, considering maximal trace semantics, we have $\mathcal{M} \models_{\max} \neg \mathbf{FG}(x \wedge y)$ and also $\mathcal{M} \not\models_{\max} \mathbf{G}x$.

Finite Executions as Infinite Traces. If one uses an infinite trace framework, some techniques can be used to encode finite traces. For instance, adding stuttering, i.e., a looping transition, in every state, allows to represent every truncated trace (by considering the truncated trace plus a loop in the last state). Figure 2d shows the corresponding transition system for our example. We can see that this adds traces that could be unexpected (depending on the system under study). Besides, it is not possible to distinguish between a trace that is truncated and a trace that models an infinite execution looping in its last state.

Another approach allows the user to reason about maximal finite traces within an infinite trace semantics. It consists in marking all the states that have no successor with a special atomic proposition (e.g., *dead*), and adding a loop to these states. As shown in Fig. 2e. It is then possible to identify *deadlocks* but all states have successor states. So, considering our example, the formula $\mathbf{G}(\neg x \rightarrow \mathbf{X}\neg x)$ would then be satisfied.

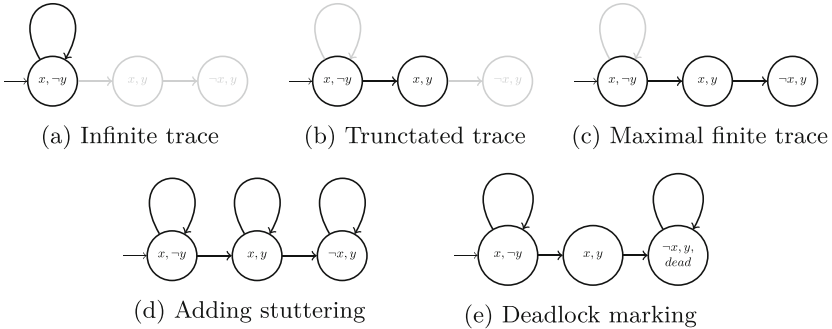


Fig. 2. Different kinds of traces

2.4 Conclusions

Among the different semantics we defined, we claim that an LTL verification tool should allow the user to choose between them, depending on the kind of system under study and on the current verification task: simulation, fast scenario finding, or property verification. This is why we included the different semantics within our solver.

3 Tatam Solver

In this section we present our solver Tatam¹ (Transition And Theory Analysis Machine), which implements the the proposed verification technique and relies on the SMT solver Z3.

3.1 Model Description

A model in Tatam is a transition system where the states are defined through the declaration of constants and variables over different possible concrete domains (also called data types in the remainder) that are commonly handled by SMT solvers such as Z3: Booleans, bounded integers, unbounded integers, real numbers. As we will see, the use of infinite data types has a consequence on the termination of some procedures.

The rest of the model is described in a usual way by an initial condition, constraints on all states (called invariant) and constraints on the transitions, which are all Boolean formulas. The atoms of these formulas refer to the state variables. The possible reasoning (variable comparison, arithmetic, call to uninterpreted functions) comes from the theories that are supported by Z3. Within constraints on the transitions, variables can be primed to denote their value in the next state (after the transition occurrence). We then define an LTL formula and ask the tool to exhibit a trace of the model that satisfies the formula. Remark that if one wants to check whether a formula φ is satisfied by all the model traces, then $\neg\varphi$ has to be specified in Tatam: either there is a trace satisfying $\neg\varphi$ (so it is not the case that φ is true for all traces), or there is no trace satisfying $\neg\varphi$ (so φ is true for all traces).

Finally, we specify the kind(s) of traces that we consider in the analysis: `infinite`, `truncated`, `finite` (meaning maximal finite) or any combination of the three kinds. For instance, if we want to consider maximal finite traces, we can specify `infinite + finite`. Additionally, we can specify bounds on the trace length: `search[1..20]` means we only search for traces of length $k \in 1..20$. If no bound is specified, the search will go on for any $k \in \mathbb{N}$ until a trace is found or a completeness threshold is reached. To check whether a completeness threshold has been reached for each new length k , we use the keyword `complete`. Notice that if there is no trace satisfying the formula and if `complete` is not specified, then the analysis obviously does not stop. Figure 3 shows the transition system presented in Sect. 2.3, with the LTL formula $\mathbf{F}\neg x$.

3.2 Resolution

Our resolution method is based on an SMT encoding of the BMC problem as presented in [10]. We present here the main ideas of the encoding of the different kinds of traces and of the treatment of the `complete` option.

¹ <https://crates.io/crates/tatam>.

```

var x, y: Bool
init I { x and not y }
trans T0 { x and not y and x' and not y' }
trans T1 { x and not y and x' and y' }
trans T2 { x and y and not x' and y' }
prop = F (not x)
search infinite + finite + complete solve

```

Fig. 3. Tatam simple example

Common Segment. In our encoding of the BMC problem, the part that is common to the different semantics consists of

- k unfoldings of the transition relation, which provides $k+1$ states S_i ($i \in 0..k$),
- and the encoding of the LTL formula, including the semantics of the LTL connectives that are included in the LTL formula.

Each state variable that is declared in Tatam gives rise to $k+1$ SMT variables (one for each trace state) in the encoding. Each transition step (S_i, S_{i+1}) is then encoded as an SMT formula $T(S_i, S_{i+1})$ over the corresponding SMT variables. Similarly, the initial conditions are encoded as an SMT formula $I(S_0)$. Given a transition system \mathcal{M} , we denote by $\llbracket \mathcal{M}, k \rrbracket$ the SMT formula encoding the k unfoldings of the transition relation:

$$\llbracket \mathcal{M}, k \rrbracket \stackrel{\text{def}}{=} I(S_0) \wedge T(S_0, S_1) \wedge \dots \wedge T(S_{k-1}, S_k)$$

We also need to encode the LTL formula φ provided by the user. To do so, an SMT variable is created for each subformula (including φ itself) in each state S_i ($i \in 0..k$). Then, we specify constraints that express the semantics of the toplevel connective of each subformula, e.g., the equivalence between $\mathbf{X}A$ in the current state and A in the next state if $\mathbf{X}A$ is a subformula of φ , the equivalence between $\mathbf{G}A$ in the current state and A and $\mathbf{G}A$ in the next state if $\mathbf{G}A$ is a subformula, etc. This encoding of the formula is denoted by $\llbracket \varphi, k \rrbracket$. For instance, let us consider $\llbracket \varphi, 2 \rrbracket$ with $\varphi = \mathbf{X}p$ (where p is a Boolean variable). In this case, the SMT variables are p_0, p_1, p_2 , which encode the variable p in states S_0, S_1, S_2 , and $(\mathbf{X}\neg p)_0, (\mathbf{X}\neg p)_1, (\mathbf{X}\neg p)_2$, which encode the formula $\mathbf{X}\neg p$. We have $\llbracket \varphi, 2 \rrbracket = (\mathbf{X}\neg p)_0 \wedge ((\mathbf{X}\neg p)_0 \leftrightarrow p_1) \wedge ((\mathbf{X}\neg p)_1 \leftrightarrow p_2) \wedge ((\mathbf{X}\neg p)_2 \leftrightarrow p_3)$.

As illustrated by Fig. 4, the part of the BMC encoding that is common to the finite and infinite semantics says nothing about the semantics of the temporal connectives in the last state (because this is precisely what differs from one semantics to the other). Therefore, the common segment is defined by

$$\llbracket \mathcal{M}, k \rrbracket \wedge \llbracket \varphi, k-1 \rrbracket$$

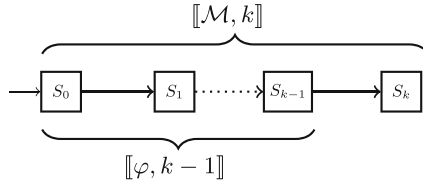


Fig. 4. Trace encoding (common segment)

Truncated. The BMC problem according to the truncated semantics is encoded as the conjunction of the common segment and a constraint $\text{trun}(k)$ that specifies the semantics of all the subformulas of φ in the last state S_k according to the LTL semantics on finite traces: any subformula $\mathbf{X}A$ is false, any subformula $\mathbf{F}A$ is equivalent to A , any subformula $\mathbf{G}A$ is also equivalent to A , etc. For instance, considering $\varphi = \mathbf{X}p$, we have $\text{trun}(2) = \neg X.p.2$.

$$\text{Truncated BMC encoding : } [[\mathcal{M}, k]] \wedge [[\varphi, k - 1]] \wedge \text{trun}(k)$$

Infinite. The case of infinite traces is encoded in the traditional way using a lasso-shape trace, as, *e.g.*, in [10]. We use a constraint $\text{loop}(k)$ that specifies the existence of a loop and the semantics of the subformulas of φ according to the infinite trace semantics and taking into account the loop in the trace.

$$\text{Infinite trace BMC encoding : } [[\mathcal{M}, k]] \wedge [[\varphi, k - 1]] \wedge \text{loop}(k)$$

Finite. The BMC encoding for the maximal finite case (simply called finite in Tatam) is more complex. For a system of length k , we search for a trace that satisfies the formula φ with the truncated semantics (providing $[[\mathcal{M}, k]] \wedge [[\varphi, k - 1]] \wedge \text{trun}(k)$ to Z3). If such a trace τ exists, we prove that it is maximal. To do so, we encode the trace τ in SMT and specify that it is extended with another step of the transition relation. If this has no solution, we have proved that τ is maximal finite. Otherwise, we search for a new solution τ of length k using the truncated semantics. If no new solution is found, we increase the number of transitions and repeat the process.

Complete. Determining whether a completeness threshold has been reached is close to the treatment of the maximal finite case. The problem here is the following: if no trace has been found until length k , we want to know whether it is worth exploring length $k + 1$. To do so, we encode the BMC problem for the length $k + 1$ considering the semantics which is chosen by the user (truncated, infinite, maximal finite or a combination) and add a constraint specifying that the last state is different from the other states. If there is no solution, then there is no need to explore length $k + 1$. If there is a solution, we need to go the length $k + 1$.

Optimization. The search for a trace that satisfies a formula proceeds by incrementing the number of transitions. Consequently, the first solution found is optimal in terms of length. However, in some cases we are looking for a trace that minimizes (or maximizes) a criterion that makes sense from the point of view of the model we are analyzing. Therefore, we've added the ability to search for a trace that optimizes a criterion that is a language expression. Moreover, when we want to optimize a criterion that refers to a variable, we need to specify which state we're talking about. In most cases, we're looking for the first or last state (or a shift) of the transition system.

To do this, we use the SMT solver's ability to find an optimal solution for a transition system of length k . When a solution is found, we add the constraint that the other solutions must improve the criterion for the following number of transitions. Thus, coupled with the `complete` semantics, the solver produces an optimal solution for a given criterion. The example presented in Sect. 5 illustrates an optimal trace search.

3.3 Conditions on the SMT Theories

Among the four main options of Tatam (`truncated`, `infinite`, `finite` and `complete`) some are only possible under some hypotheses on the SMT domains of the model. For instance, if the comparison between states is needed, which is the case in the encoding of lasso-shape traces (`infinite`) and in the more complex procedures involved for `finite` and `complete` options, then parameters of variable functions need to be declared with a finite domain. Only the most simple option (`truncated`) imposes no constraints on the domains. The most demanding option is the `finite` one because it requires to iterate over all the solutions that are returned by the SMT solver. This is why this option requires constants and variables to be declared over finite domains in order to guarantee the termination of the solving. On the other hand, the `complete` option does not require constants to be declared over finite domains because there is no need to ask for a solution with a different valuation of constants.

Note that for `finite` and `complete` options, the procedures do not directly specify an SMT formula, unlike the `truncated` and `infinite` options. As briefly explained above, the resolution in these cases involves more complex algorithms that do multiple calls to SMT solvers. In some cases (such as an infinite domain for variables or for parameters of variable functions) these algorithms do not terminate. In such situations, we say that the problem is not expressible in SMT. In case a problem is expressible, it is still possible that the SMT solving does not terminate. In table 1, we call termination the fact that the problem is expressible and the SMT solving necessarily terminates.

Table 1 outlines various conditions on the constant and variable domains for the different options of Tatam. Of course, if multiple options are selected, the strongest condition applies. F . denotes finite domains, I . denotes infinite

domains; *cst fun p.* (resp. *var fun p.*) refers to constant (resp. variable) function parameters. Note that only the **truncated** option allows variable functions with infinite parameters. Also note that for **complete** and **finite** options, the Tatam problem is expressible in SMT with variables over infinite domains but the termination is only guaranteed if all variables are declared on a finite domain.

Table 1. Conditions on the domain to ensure termination

	constant	variable	cst fun p.	var fun p.
	F.I.	F.I.	F.I.	F.I.
truncated	✓✓	✓✓	✓✓	✓✓
infinite	✓✓	✓✓	✓✓	✓
finite expressible	✓✓	✓✓	✓✓	✓
finite terminates	✓	✓	✓✓	✓
complete expressible	✓✓	✓✓	✓✓	✓
complete terminates	✓✓	✓	✓✓	✓

4 Benchmarking

In this section, we present two simple benchmarks to assess the practicality of our approach, rather than perfectly characterizing our tool against existing ones. We compare the computation times of our prototype with the reference tools NuSMV and nuXmv, which implement various resolution algorithms:

- BDD-based model checking, which is only applicable to finite domains, and performs complete model checking,
- BMC (Bounded Model Checking), which is also applicable to finite domains, and does not perform complete model checking: if no trace (or counterexample) is found for a trace length k , the tool cannot conclude,
- IC3 “bounded”, which performs K-Liveness translation from liveness to safety and applies IC3 algorithm using a SAT encoding (applicable to finite domains),
- IC3 “unbounded”, which is applicable to infinite domain variables, and performs similarly to IC3 “bounded” but relies on an SMT encoding.

One way to see our approach is that we extend NuSMV/nuXmv implementation of BMC by handling infinite types and providing a completeness threshold detection, which provides complete model checking.

Since the other tools do not handle maximal finite traces and some do not handle truncated traces, we limit ourselves to the infinite trace case for the comparison. To test the performance, we increase the problem size and measure the solution time

1. to find a *trace* satisfying an LTL formula,
2. and to prove the absence of traces satisfying another LTL formula (what we call *proof* in the remainder of the article).

In practice, we used a formula which is satisfied by all traces for the first case and its negation for the second case. For BMC, a sufficiently large bound on transitions ensures complete coverage. All computations are performed on a Intel(R) Xeon(R) W-11955M CPU @ 2.60GHz and 64GB of memory, with a time limit of 10 min per iteration.

4.1 Leader Election

The first example we present concerns a leader election protocol [11], which assumes a ring network of processes (nodes) with unique, comparable identifiers. Each node can communicate with its successor in the ring by sending its own identifier, or the identifier it received from its predecessor if it is greater than its own identifier. If a node receives its own identifier, it considers itself as the elected leader. We have modeled this system using Boolean variables only. The LTL formula we checked says that there will eventually be a leader, or more precisely: eventually one of the nodes will receive its own identifier. It is important to note that, in this example, unbounded solution of IC3 is not possible because the problem contains only Boolean variables.

Figure 7 shows the computation time of the different solvers on the y-axis and the size of the ring on the x-axis. We observe that the use of BDD is significantly more efficient than the other algorithms. Our solver behaves similarly to IC3 when it comes to finding a trace. However, its performance deteriorates when searching for a proof (no trace), with only BMC performing worse in this case.

4.2 Arithmetic Sequence and Series

We designed our tool to handle problems involving unbounded variables, arithmetic, and LTL formulas. However, most benchmarks in the literature focus on discrete problems [12–16]. Our second example involves computing an arithmetic sequence and an arithmetic series. We define the transition system for any arithmetic sequence and series, adding the constraint that the series value must be reached after a given number of iterations. The solver must find the initial parameters and all values of the sequence and series. Figures 5 and 6 represent the Tatam and SMV models, respectively, for iteration 10. To get an infinite trace, we add a transition at the beginning of the series. Finally, the LTL formula we check is $\mathbf{GF}(n = 0)$ and its negation, depending on whether we need a trace or a proof.

```

cst r: Int
var n, u, s: Int
init I {
  n = 0 and r > 0 and
  u = 0 and s = 0
}
trans step {
  if n = 10 and s = 165 then
    n' = 0 and
    u' = 0 and
    s' = 0
  else
    n' = n + 1 and
    u' = u + r and
    s' = s + u'
  end
}

```

Fig. 5. Tatom

```

JUSTICE TRUE;
FROZENVAR r: integer;
VAR n: integer;
VAR u: integer;
VAR s: integer;
INIT
  n = 0 & r > 0 &
  u = 0 & s = 0 ;
TRANS
(
  n = 10 & s = 165 &
  next(n) = 0 &
  next(u) = 0 &
  next(s) = 0 ) |
(
  ! (n = 10 & s = 165) &
  next(n) = n + 1 &
  next(u) = u + r &
  next(s) = s + next(u) );

```

Fig. 6. SMV

Note that we added bounds to the variable values to address the *bounded* solvers and for the *proof* search. Figure 8 shows the computation time of the different solvers on the y-axis and the problem size on the x-axis. This example demonstrates the advantage of using SMT solvers for mathematical transition functions to find a trace. We also observe that our solver is more efficient than the IC3/SMT solution for computing traces. The algorithm we presented, which indicates the termination of the BMC and enables proofs, is also very efficient at computing proofs when others struggle.

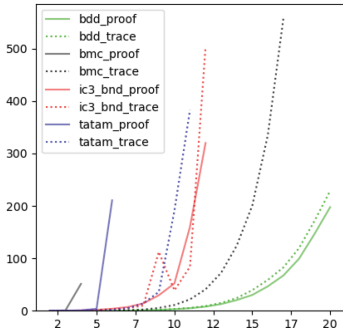


Fig. 7. Leader election benchmark

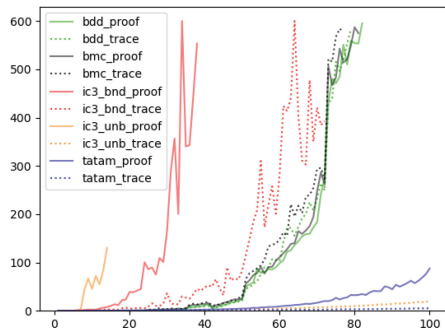


Fig. 8. Arithmetic series benchmark

5 Example

In this section, we present a simple example to demonstrate the value of the different semantics that we introduced earlier. This example is inspired by several

analyses that were performed on real robotic systems. We want to model a small robot that makes deliveries from a factory to a client. When a delivery is ready at the factory, the robot must load boxes, go to the client, and unload the box. If the delivery is incomplete, it must return to the factory and start again until all the boxes have been delivered. The robot consumes fuel during its movements. The fuel consumption depends on the number of transported boxes. The capacity of the robot (the number of boxes it can transport at a time) is 5, and the number of produced boxes is 3. The robot tank, which is full initially, has a size of 30. It consumes 1 for a trip from the client to the factory. From the factory to the client, it consumes 10 if it is carrying 2 or less of boxes, and 25 otherwise.

5.1 Transition System

The system state is described by five variables: representing the state of the robot, its tank, and the number of boxes in the factory, on the robot, and at the client. Figure 9 shows an extract from the variable declaration, the definition of the initial state and an invariant.

There are six transitions: **loading**, corresponding to the loading of at least one box on the robot; **unloading**, the unloading of the cases present on the robot once it has arrived at the client; **start_to_client** representing the departure to the client; **finish_to_client** representing the arrival at the client; **start_to_factory** and **finish_to_factory** for the movement to the factory. Figure 10 shows the definition of the transitions **start_to_client** and **finish_to_factory**.

```

enum RobotState = {
    AtFactory, AtClient,
    ToFactory, ToClient
}

var robot_state: RobotState
var robot_fuel: Int
var factory_boxes: Int
var robot_boxes: Int
var client_boxes: Int

init Init_Var {
    factory_boxes = 3 and
    robot_boxes = 0 and
    ...
}
inv Inv_Fuel {
    robot_fuel >= 0
    ...
}

```

Fig. 9. Variables, Init, Inv

```

trans start_to_client {
    robot_state = AtFactory
    and robot_boxes > 0 and
    |robot_state| (
        robot_state' = ToClient
    )
}

trans finish_to_client {
    robot_state = ToClient and
    |robot_state, robot_fuel| (
        robot_state' = AtClient
        and
        robot_fuel' = robot_fuel -
        if robot_boxes <= 2 then
            10
        else
            25
        end
    )
}

```

Fig. 10. To client transitions

5.2 Properties

Now, let us study some typical expected properties, with the corresponding semantics for each of them.

Complete Delivery. The first property we want to check is whether it is possible to deliver to the client. In this case, we look for any trace, finite or infinite, that reaches a state where the delivery is done. Thus, it is enough to ask the solver to find a trace satisfying $F(\text{client_boxes} = \text{produced_boxes})$ according to the truncated semantics. Our solver finds a trace of length 4 in which the robot loads all the boxes, goes to the client, and delivers all the boxes. When it arrives at the client, the quantity of fuel in its tank is 5. Since our solver works by incrementing the length the candidate trace, the proposed solution necessarily has the minimum length.

Break Down. Next, we want to find out if it is possible for the robot to run out of gas on its way to the client. Notice that in our model, this does not necessarily correspond to a state where the value of the tank is 0. Indeed, if the value of the tank is less than what is needed to reach the client, then the transition `finish_to_client` cannot occur, so the value of the tank is not decremented. A possibility to specify this situation consists in expressing that the robot reaches the state `ToClient` and does not leave this state: $FG(\text{robot_state} = \text{ToClient})$.

If we search for a trace with the *infinite* semantics, the solver will not find a trace, because there is no infinite scenario that satisfies this formula. And yet it can happen!

If we use truncated semantics, the solver finds a trace of length 2 that satisfies the formula. The truncated trace that stops at the first move indeed does satisfy the formula. However, this trace does not correspond to a complete scenario (the robot can keep moving) and therefore does not match what we are looking for.

Finally, when we look for a trace with the maximal *finite* semantics, the solver shows a trace of length 14 in which the robot runs out of fuel. This happens when the robot picks up boxes one at a time, so on its first trip to the client it has 20 left in its tank, on its second trip it has 9 left, and on its last trip to the client it has 8 left and runs out of fuel because it needs 10 to get there.

Optimal Delivery. Now, we want to find the best way to deliver to the client, the one that minimizes fuel consumption. To do this, we can use our solver optimization search and tell it that we are looking for the trace that maximizes the amount of fuel in the final state of the trace. The solver finds a trace of length 10 in which the amount of fuel is 9. This solution is optimal from the point of view of the criterion (in this case, remaining fuel). Figure 11 shows the different searches performed for this example².

² https://github.com/DavidD12/tatam/tree/main/files/icfem_2024.

```

prop = F(client_boxes = produced_boxes)
search truncated solve

prop = F(G(robot_state = ToClient))
search finite solve

prop = F(client_boxes = produced_boxes)
search truncated + complete
      maximize (robot_fuel at last) until robot_fuel_capacity

```

Fig. 11. Example searches

6 Conclusion

We proposed a way to perform LTL model checking for finite and infinite traces. Our method is based on an SMT encoding and on classical BMC. The use of an SMT solver enables the use of concrete variable domains, such as bounded and infinite integers and real numbers. We also used a simple criterion to detect that a completeness threshold has been reached for the length of traces. This provides a complete model checking procedure, which necessarily terminates in case the concrete variable domains are finite. We experimented our tool and compared it to NuSMV and nuXmv.

References

1. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI 2013, pp. 854–860. AAAI Press (2013)
2. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Campenhout, D.V.: Reasoning with temporal logic on truncated paths. In: International Conference on Computer Aided Verification (2003)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Specification. Springer, Heidelberg (1991). <https://doi.org/10.1007/978-1-4612-0931-7>
5. Cimatti, A., et al.: NuSMV 2: an opensource tool for symbolic model checking. In: CAV, Denmark, Copenhagen (2002)
6. Cavada, R., et al.: The NUXMV symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
7. Macedo, N., Brunel, J., Chemouil, D., Cunha, A.: Pardinus: a temporal relational model finder. *J. Autom. Reason.* **66**(4), 861–904 (2022)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, revised edn. MIT Press, Cambridge (2012)
9. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)

10. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. *Logical Methods Comput. Sci.* **2** (2006)
11. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM* **22**(5), 281–283 (1979)
12. Pakonen, A.: Model-checking infinite-state nuclear safety I&C systems with nuXmv. In: 19th IEEE International Conference on Industrial Informatics, INDIN 2021, Palma de Mallorca, Spain, 21–23 July 2021, pp. 1–6. IEEE (2021)
13. Schuppan, V., Darmawan, L.: Evaluating LTL satisfiability solvers. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 397–413. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_28
14. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. *Int. J. Softw. Tools Technol. Transf.* **12**(2), 123–137 (2010)
15. Wulf, M.D. , Doyen, L., Maquet, N., Raskin, J.: Antichains: alternative algorithms for LTL satisfiability and model-checking. In: TACAS, Budapest (2008)
16. Li, J., Pu, G., Zhang, L., Vardi, M.Y., He, J.: Fast LTL satisfiability checking by SAT solvers. CoRR [arxiv:1401.5677](https://arxiv.org/abs/1401.5677) (2014)



Model Checking Concurrency in Smart Contracts with a Case Study of Safe Remote Purchase

Yisong Yu^{1,2}, Naipeng Dong^{3(✉)}, Zhe Hou⁴, and Jin Song Dong²

¹ Ningbo University, Ningbo 315211, Zhejiang, China
e0792458@u.nus.edu

² National University of Singapore, 21 Lower Kent Ridge Road,
Singapore 119077, Singapore
dcsdjs@nus.edu.sg

³ The University of Queensland, St. Lucia, Brisbane, QLD 4072, Australia
n.dong@uq.edu.au

⁴ Griffith University, 170 Kessels Road, Nathan, QLD 4111, Australia
z.hou@griffith.edu.au

Abstract. Blockchain technology has evolved beyond its initial role in supporting cryptocurrencies like Bitcoin, with Ethereum introducing smart contracts for decentralised applications in various domains. However, ensuring the safety and security of smart contracts remains a critical challenge, particularly concerning concurrency issues. This is of paramount importance because the smart contract ecosystem is concurrent by nature as its underlying blockchain is decentralised, and the concurrency-related vulnerabilities within smart contracts have resulted in substantial financial losses. We observe that in the literature, concurrency is handled with two strong assumptions, leading to either undetected attacks or false alarms. Taking the Safe Remote Purchase smart contract as a case study, we investigated the root causes and introduced a novel method that incorporates blockchain-specific characteristics into the verification process. Our contributions include a formal framework, an automated model generator, and a compelling case study that illustrates a reduction in false attacks, thus advancing the state of smart contract security in blockchain ecosystems. The formal models and the framework generator are available online at <https://github.com/FormalVerificationBlockchain/Concurrency>.

Keywords: Smart Contracts · Concurrency · Formal Verification · Blockchain

1 Introduction

Blockchain was introduced as the fundamental technology for supporting the well-known cryptocurrency—Bitcoin in 2008. Later, Ethereum [26], known as Blockchain 2.0, aims to provide an open and decentralised platform for general-purpose computing through the introduction of a groundbreaking concept called *smart contract*. It enables the development of decentralised applications across various domains, from finance to supply chain management.

Smart contracts, which are Quasi-Turing-Complete programs running on top of a blockchain, have a unique feature that, once deployed, it is costly to change them (*i.e.*, patching is nearly infeasible) due to the irreversible feature of the underlying blockchain that stores the programs. Therefore, verifying the correctness of smart contracts before deploying them to a blockchain is crucial. Time and again, incidents have led to huge financial losses due to bugs, breaches and logic flaws in smart contracts—*e.g.*, the well-known DAO attack [18], Parity Multisig Wallet attack [8], and the King of the Ether Throne attack [1]. A variety of techniques have been developed to verify the safety and security of smart contracts and their applications in the past several years, including design patterns [27], informal vulnerability detection [3] and formal verification approaches [24]. Formal verification stands out because many smart contract applications are safety-critical *e.g.*, supply chain, finance, and medical services, and formal verification provides rigorous proof contrary to other approaches.

However, most of the formal verification techniques take smart contracts independently, isolating them from other participants, and consider their semantics as sequential programs, as pointed out by Sergey and Hobor [17]. Such methods overlook complex interactions and can lead to security risks. Luu et al. [13] first identified security issues caused by the network participants, particularly pointing out that the transaction orders are non-deterministic, which can be utilised by malicious users to gain benefits, known as the Transaction-Ordering Dependence (TOD) problem. A subsequent study [17] generalised the problem to be the concurrency issues and introduced two additional types of concurrency scenarios: *Multitasking* and *Multi-Transactional Tasks*, by drawing analogies to the concurrent objects in shared memory in the well-studied concurrency field named as *concurrent-objects-as-contracts*.

To address these concurrency issues, Luu et al. [13] developed a tool, OYENTE, to detect TOD in smart contracts based on symbolic execution. Although rigorous, OYENTE still suffers from a large number of both falsely reported TOD and missed TOD cases because its execution environment of Ethereum is not fully simulated [15]. This observation was also noted by Sergey and Hobor [17], highlighting that verification of smart contracts requires modelling the interactions with other components. While Sergey and Hobor [17] demonstrated the concurrency problems using example smart contracts and their vulnerabilities, they did not delve into verification approaches. Building upon this approach—concurrent-objects-as-contracts, a later work [16] took the first step towards formalising concurrency. Taking the Safe Remote Purchase smart contract as a case study, the work [16] studied its source code, modelled the smart contract using the Communicating Sequence Processes (CSP), proposed an attack model, and verified the existence of a concurrency attack using the model checker FDR.

Upon examining the model and verification in [16], we identified two limitations: 1) The attack is not automatically detected. The attack model represented as a trace embedding of the identified vulnerability, requires the analysts to have prior knowledge of vulnerability exploration and analyse it manually. 2) More significantly, naively adopting the concurrent-objects-as-contracts approach results in *false attacks*. We show that the attack reported in [16] is, in fact, a false alarm in the context of the blockchain ecosystem.

The first limitation can be straightforwardly addressed by modifying the query, given that most model checkers are capable of automatically detecting concurrency issues, as shown in Sect. 4. The second limitation stems from a misunderstanding of the blockchain foundation, which is more challenging. The blockchain foundation cannot be treated as a single machine, as demonstrated in the above works; otherwise, there will be concurrency vulnerabilities. On the other hand, the transactions are not executed in a completely interleaving manner; otherwise, there will be false attacks. To strike a balance between these two ends of the spectrum of modelling approaches, we show that including a blockchain component in the modelling, even an abstracted one, would address the limitation. Moreover, we developed a general method by proposing a verification framework that facilitates verification beyond the case study.

The contributions of this work can be summarised as follows:

- We identified false attacks within existing approaches to verify the concurrency of smart contracts and have investigated their root causes, namely the absence of a correct blockchain execution model.
- We proposed an approach that addresses the identified issue by modelling the environment participants, including the blockchain and the external actors.
- We generalised the approach and developed a script to automatically generate formal models that facilitate the verification of smart contracts.

2 Concurrency Issues

EVM is a *single-threaded* state machine which cannot process instructions in parallel (except parallel EVM *e.g.*, [25]), suggesting that smart contract methods can be deemed, in a traditional programming context, as *sequential programs*. This is deceptive and misleading, given concurrent behaviours *e.g.*, reentrancy and recursive calls, can still be observed, if investigating it from a different perspective—at the level of blockchain ecosystem. More specifically, the following complications have been observed and demonstrated in the literature:

Transaction-Ordering Dependence. Although each transaction is always deterministic, *non-determinism* may still arise from races between transactions *i.e.*, out-of-order executions from the perspective of an external actor, leading to distinct outcomes. For instance, malicious external users could front run a transaction by providing higher fee (front-running attack), and malicious miners could reorder transactions to gain profits (Miner Extractable Value attack) [10].

Multitasking. Calling other contracts or oracles, dictates an explicit “yield” or “relinquishment” of control that will not be handed back to the caller until the callee contract returns. During this time, several things can go wrong: The callee contract can run whatever code it likes or even call other external contracts to engage in unexpected activities without getting interrupted. The caller may be malicious *e.g.*, in the DAO attack the caller re-enters the smart contract to draw extra fund. The callee’s input arguments and return values are passed using volatile memory space that may be compromised.

Multi-transactional Tasks. The contract logic of some programming tasks, which should conceptually be grouped into a single logical transactional entity, sometimes needs to be separated across multiple physical blockchain transactions. Between these transactions, other transactions whose method invocation may involve manipulation of the state unanticipated by the original proposing actor, can take place, resulting in true concurrent behaviours.

```

1  pragma solidity ^0.8.4;
2  contract Counter {
3      address public id; uint private count;
4      constructor() payable {
5          id = msg.sender; count = msg.value; } }
6      function get() returns (uint) { return count; }
7      function set() returns (uint) {
8          uint oldCount = count; count = msg.value;
9          msg.sender.transfer(oldCount);
10         return oldCount; }

```

Fig. 1. Example Smart Contract Code with Concurrency Issues.

Example 1 (Smart Contract with Multi-transactional Issues). The smart contract in Fig. 1 has three functions: a **constructor** function that imitates the smart contract by assigning values to *id* and *count*, a **get** function that reads the value of *count*, and a **set** function that updates the *count* value and transfers fund worth the previous *count* value (*i.e.*, *oldCount*) to the caller.

Imagine the following scenario: Alice wants to update the value *count* by calling the function **set**, and before doing so, she first calls the function **get** to check how much fund she would receive *i.e.*, the current value of *count*. Even the order of Alice’s transactions (**get** followed by **set**) is correctly preserved, other problem arises, as there is no guarantee that other function invocation requests won’t be scheduled in between Alice’s two transactions. For example, Bob may have called the **set** in between, which resets the *count* value. In this case, Alice would get a different amount of fund (Bob’s updated value of *count*) and Bob’s set value of *count* will be re-written by Alice’s update.

Multi-transactional behaviours are the problem that the concurrent-objects-as-contracts approach claims to be able to address. Note that the above concurrency issues in Example 1 is a true attack, which can be confirmed by both the concurrent-objects-as-contracts approach and our formal verification framework (detailed in Sect. 5.2). In contrast, in the subsequent section, we show another smart contract as our running example to demonstrate false attacks occur if using the concurrent-objects-as-contracts approach naively.

3 The Safe Remote Purchase Smart Contract

We analyse the same smart contract—the Safe Remote Purchase—as in [16] that uses concurrent-objects-as-contracts approach.

3.1 The Case Study Smart Contract

The smart contract aims to help mutually distrusting parties, for example, a seller and a buyer in the simplest configuration, to achieve a safe and reliable



Fig. 2. Flowchart of Safe Remote Purchase Transaction [16].

```

1  contract Purchase {
2      uint public value;
3      address payable public seller, buyer;
4      enum State { Created, Locked, Release,
5          Inactive }
6      error OnlyBuyer(), OnlySeller(), InvalidState
7          (, ValueNotEven());
8      modifier condition(bool condition_) {
9          require(condition_); }
10     modifier onlyBuyer() {
11         if (msg.sender != buyer)
12             revert OnlyBuyer(); }
13     modifier onlySeller() {
14         if (msg.sender != seller)
15             revert OnlySeller(); }
16     modifier inState(State state_) {
17         if (state != state_)
18             revert InvalidState(); }
19     event Aborted(), PurchaseConfirmed(),
20         ItemReceived();
21     constructor() payable {
22         seller = payable(msg.sender);
23         value = msg.value / 2;
24         if ((2 * value) != msg.value)
25             revert ValueNotEven(); }
26     function abort()
27     external onlySeller inState(State.Created) {
28         emit Aborted(); state = State.Inactive;
29         seller.transfer(address(this).balance); }
30     function confirmPurchased()
31     external inState(State.Created)
32     condition(msg.value == (2 * value))
33     payable { emit PurchaseConfirmed();
34         buyer = payable(msg.sender);
35         state = State.Locked; }
36     function confirmReceived()
37     external onlyBuyer inState(State.Locked) {
38         emit ItemReceived();
39         state = State.Release;
40         buyer.transfer(value);
41         seller.transfer(address(this).balance); } }

```

Fig. 3. Solidity Code of Safe Remote Purchase [19].

purchase transaction on a decentralised e-commerce platform built on top of Ethereum, by resolving the issue with the confirmation of the shipment and the acknowledgement of the confirmation prior to settlement (Fig. 2). The general strategy proposed by this contract is to enforce a guaranty/deposit of twice the value of the item to be transferred into the contract account as escrow on both parties of a remote purchase transaction. The deposit stays locked until the buyer confirms the receipt of the goods, triggering a refund of the appropriate amount to both parties, *i.e.*, the value equivalent to half of the deposit to the buyer and three times the value equivalent to the guaranty plus the value to the seller.

The solidity code of the smart contract is shown in Fig. 3. The constructor function (lines 19–23) is called by the seller to create a contract and pay the guaranty. Then, the buyer can initiate a purchase by calling the function `confirmPurchased` with the payment of a deposit as a parameter (“`msg.value`”). Then, the smart contract locks the payment (line 33). After the seller delivers the purchased item, the buyer invokes the function `confirmReceived` to notify the contract that the item has been received, where the smart contract refunds the seller and buyer *i.e.*, releasing the lock and paying the corresponding funds to the seller and buyer (line 37–39). In this process, the seller is allowed to abort the contract by calling the function `abort`. The abort function sends the balance of the contract to the seller (line 27).

3.2 Formal Analysis of Concurrency in Existing Work

Wang et al. [16] showed an attack trace when performing model checking using the concurrent-objects-as-contracts approach, *i.e.*, analogous concurrency in smart contracts to racing problems in traditional concurrent programs. The vulnerability discovered lies in-between the execution of line 28 in Fig. 3, which

sends deposit to the contract account, and line 33, which locks the payment. From a concurrency perspective, in this specified period, the seller is allowed to call the `abort` function, which changes the state of the purchase to be aborted and the total balance of the contract's account, containing the guaranty/deposit of both seller and buyer, will be transferred to the seller's account.

Critiques. We identified two limitations of the above formal modelling and verification: First, differing from traditional model checking approaches, this model came up with an attacker trace in advance as the input to the FDR model checker to only confirm the existence of such a counterexample trace in all possible executions of the modelled smart contract. The attack trace is an artefact usually not available beforehand, as no one would be able to know what kind of attacks can be launched against some vulnerabilities that may or may not even exist in his/her built contract. Second, this model lacks environmental considerations, such as expected and unexpected user interactions, *e.g.*, possible behaviours of a seller/buyer through the provided interfaces (those demonstrated in Fig. 2).

4 Analysing Existing Verification Approach in PAT

The first critique can be straightforwardly addressed, as the attack trace can be automatically identified by many existing model checkers. We illustrate it by mimicking the modelling and verification of [16] using the CSP # modelling language supported by the model checker PAT (Process Analysis Toolkit) [21]. The reason for choosing another model checker is that the approach used in [16] lacks support for state variables and many other common programming constructs for flow controls, which hinders our goal of developing a general framework (detailed in Sect. 5), while PAT enables us to develop external libraries and functions.

4.1 Introduction to CSP# and PAT

PAT is a self-contained framework that supports the composing, simulating, and reasoning of concurrent systems using various model-checking techniques. More importantly, PAT has been developed as a generic framework, which can be easily extended to support new modules and frameworks. The input language of PAT is CSP#, whose syntax is shown in Definition 1.

Definition 1 (Syntax of Processes in CSP#).

$$P, Q ::= \text{Stop} \mid \text{Skip} \mid e \rightarrow P \mid e\{\text{prog}\} \rightarrow P \mid c\text{exp} \rightarrow P \mid c?x \rightarrow P \mid \\ P \parallel Q \mid \text{if}(b)\{P\} \text{ else } \{Q\} \mid \text{ifa}(b)\{P\} \text{ else } \{Q\} \mid P; Q \mid P \parallel Q \mid P \parallel\parallel Q$$

Stop and *Skip* are processes denoting inaction and termination, respectively. Process $e \rightarrow P$ engages in an atomic event e first and then behaves as process P . The event is allowed to attach an atomically executed program, denoted as $e\{\text{prog}\} \rightarrow P$. Channel communication is supported: $c\text{exp} \rightarrow P$ denotes sending exp over channel c , while $c?x \rightarrow P$ denotes reading from channel c and

referring to the message as x . Two types of choices are supported: $P||Q$ denotes unconditional choice, and $if(b)\{P\} else \{Q\}$ is conditional branching, where b is a boolean expression. $ifa(b)\{P\} else \{Q\}$ is a variation of conditional branching that performs the condition checking and first operation of P/Q together. There are three types of process relations: Process $P; Q$ behaves as P until P terminates and then behaves as Q . The parallel composition of two processes is written as $P||Q$, where P and Q may communicate via multi-party event synchronisation. If P and Q only communicate through variables, then it is written as $P|||Q$.

4.2 Mimicking the Modelling of the Existing Work

To demonstrate the capability of PAT, we first faithfully replicate the model in [16] into a corresponding CSP# model with all required modifications to maximise the semantic equivalence between them. That is, we expect to see the same result as in the previous work, *which may not be correct*. Essentially, each function in Fig. 3 is modelled as a process in CSP#, and the smart contract is the interleaving of the functions, as shown in Fig. 4. The modelling is straightforward: the model of function constructor is in line 8–12 in Fig. 4; the model of abort is in line 13–19; confirmPurchased in line 21–28; and confirmReceived in line 29–33; line 34 is the composed smart contract.

```

1  enum {CONTRACT, SELLER, BUYER};
2  enum {NULL, CREATED, LOCKED, RELEASE,
3     INACTIVE};
4  channel constructor 1;
5  channel buyer 1;
6  channel access 1;
7  var guaranty = -1;
8  var state = NULL;
9  Constructor() = msg_value ->
10 ifa (msg_value % 2 == 0) {
11     seller.guaranty_eth_msg_value{guaranty
12     = msg_value;} -> state_created{state
13     = CREATED;} -> Purchase()
14 } else { warning -> Skip };
15 Abort() =
16 access?object ->
17 ifa (object == SELLER) {
18     ifa(state == CREATED) { abort ->
19     state_aborted{state = INACTIVE;} ->
20     eth_balance_seller_overall -> Skip
21 } else { warning -> Skip }
22 } else { warning -> Skip };
23
24 ConfirmPurchased() = access?object ->
25 ifa (object == BUYER) { ifa (state == CREATED) {
26     buyer?deposit -> ifa (deposit == guaranty) {
27     purchase_confirmed ->
28     state_locked{state = LOCKED;} -> Skip
29 } else { warning -> Skip }
30 } else { warning -> Skip }
31 } else { warning -> Skip };
32 ConfirmReceived() = access?object ->
33 ifa (object == BUYER) { ifa (state == LOCKED) {
34     item_received -> state_inactive{state = RELEASE;} -> Skip
35 } else { warning -> Skip }
36 } else { warning -> Skip };
37 Purchase() = Abort() ||| ConfirmPurchased() ||| ConfirmReceived()
38
39 Deployer(guaranty_value) = constructor!guaranty_value -> Skip;
40 Seller() = access!SELLER -> Skip;
41 Buyer(deposit_value) = access!BUYER -> buyer!deposit_value ->
42 access!BUYER -> Skip;
43 BlockchainSystem() = Constructor() || (Deployer(10); (Seller() |
44 || Buyer(10)));
45 #assert BlockchainSystem() |= [] (purchase_confirmed -> !<
46 abort);
    
```

Fig. 4. Replicated CSP# Model Code Snippets of Safe Remote Purchase [16].

To address the second critique (cf. Sect. 3.2), we integrate additional processes (the processes in line 35–38 in Fig. 4) to model the behaviours of external actors—the seller (deployer) and the buyer, to make this model complete. The external actors use the channel constructs to communicate with the smart contract functions, modelling the invocation of the functions in real-world scenarios. The input operations in the channel constructs must have the corresponding output operations to be specified in order to prevent a deadlocked scenario; therefore, we correspondingly add channel receive at the beginning of each function (*i.e.*, line 9, 14, 21 and 29 respectively).

We then formalised concurrency as assertions using the supported LTL (Linear Temporal Logic). The assertion (line 39 in Fig. 4) captures the interference

Table 1. Verification Result of the Replicated CSP# Model.

Assertion	BlockchainSystem() $\text{---} = \square(\text{purchase_confirmed} \rightarrow !\langle \rangle \text{abort})$
Counterexample	init \rightarrow constructor!10 \rightarrow access!BUYER \rightarrow buyer!10 \rightarrow constructor?10 \rightarrow seller_guaranty_eth_msg_value \rightarrow state_created \rightarrow access?BUYER \rightarrow access!BUYER \rightarrow warning \rightarrow access?BUYER \rightarrow access!SELLER \rightarrow buyer?10 \rightarrow purchase_confirmed \rightarrow access?SELLER \rightarrow abort

between the confirming purchase and aborting by querying the statement—if the purchase is confirmed, then the seller would not abort.

Verification Result Evaluations. Running this CSP# model in the PAT model checker produces the verification result as shown in Table 1 with a counterexample trace given as the proof of the invalidity of the assertion that states the safety property of this contract, resembling the attacker model trace presented in the original literature. Although not exactly identical, it captures the same attack trace where the event `abort` (line 13 in Fig. 4) is preceded chronologically by the event `purchase_confirmed` (line 24). As a consequence, the whole smart contract balance (comprised of `guaranty/deposit` from both buyer and seller) is transferred from the contract address to the seller address in the function `abort` immediately after the deposit transfer from the buyer address to the contract address, but before the state transition (from `CREATED` to `LOCKED` in line 25) operation gets performed in the function `confirmPurchased` (to bypass the pre-condition check of the function `abort`). This reveals a potential vulnerability inherent in the original contract design, which may be exploited by a malicious seller to successfully steal the deposit of the buyer if they can somehow manipulate the execution to enforce this particular sequence of executions.

Modelling Strategy Critiques. However, the above-described attack, identified in [16] and our above verification, in reality, is *not feasible* given the underlying execution model of EVM where a transaction that encapsulates a contract function invocation is an atomic operation that cannot be divided into several chunks for executions, and thus, a contract function whose execution cannot be interleaved with other functions is either completely done or not at all, implying that a *false positive* result is found based on this wrong assumption.

4.3 An Analysis of Issues in Existing Methods

The above false positive originates from a wrong interpretation of the execution model of EVM. In reality, the function `abort` can never get its turn for execution until the other function `confirmPurchased` fully finishes in the example contract. It is an incorrect use of the interleaving operator to compose the three possible actions to be performed by an external participant, corresponding to the processes `Abort`, `ConfirmPurchased`, and `ConfirmReceived` in Fig. 4.

In this particular case, there is an easy fix *i.e.*, replacing the interleaving of the three processes (`Abort`, `ConfirmPurchased` and `ConfirmReceived`) with unconditional choices of all possible permutations of these three constituent processes

sequentially composed with each other. In this way, only one single function can be executed sequentially until termination without being interrupted/preempted by other functions. However, writing this kind of long process definition spanning over more than five lines is tedious, error-prone and unsustainable. Given that there are in total $n!$ permutations for n constituent processes, hinting that it will soon become impractical to write them by hand when n becomes larger as it demands $O(n!)$ time to verify the result, making the modelling alone is an NP-hard problem already, which suffers from combinatorial explosion, before we even start tackling the state explosion issue that we are likely to encounter during the verification.

In addition, the model (in [16] and thus the same in the above model) does not handle the case of subsequent attempts of function invocations after the created purchase order has been finalised, *i.e.*, the processes are only executed once. The likelihood of the functions `abort`, `confirmPurchased`, and `confirmReceived` being invoked multiple times in arbitrary order due to the contract's immortality once deployed (assuming no *self-destruct* function) can render verification results flawed. Without accurately reflecting these non-terminating behaviours in the model, the state space may not cover all possible intricate interactions, potentially leading to unexpected side effects and false positive vulnerabilities. This issue can be resolved by making these processes recursive. Similarly, naive recursion without the correct execution model of the underlying EVM over-approximates the state space since transactions in a block are executed after the transactions in the previous blocks, prompting the development of a general framework to reduce false positives in verification results.

5 Our Approach

The above analysis indicates that the challenges can be addressed by modelling the correct interpretation of the execution model of EVM. Instead of adding a process of the EVM merely working for this case study, we aim to tackle the problem in general, as we observe that there is a clear line of demarcation drawn between the *control logic* (the underlying blockchain) and the *business logic* (the smart contract functions) being common to the modelling of any smart contract, whether already deployed or yet to be developed. This inspired us to develop a modelling approach that can be extended into a standard practice in the form of a universal framework that can be easily and effectively exploited by an average smart contract verification engineer.

In a nutshell, we abstract away unnecessary details in the *control logic* and separate it from the actual *business logic*, which is unique and thus must be independently specified and tailored by the model engineer for each target smart contract so that the control logic common to all contract model specifications can be automatically generated by our framework to address the main challenge of misunderstanding issues, which eventually ties back to a reduction in the false positive rate of the verification results and in turn becomes an improvement in the precision of the verification outcomes in general while accounting


```

1  #include "blockchain_framework.csp";
2  #define INITIAL_VALUE 0;
3  #define ITEM_PRICE_VALUE 5;
4  enum {CONTRACT_ADDRESS, SELLER_ADDRESS,
5        BUYER_ADDRESS};
6  enum {NULL_STATE, CREATED_STATE, LOCKED_STATE,
7        RELEASE_STATE, INACTIVE_STATE};
8  channel constructor 0;
9  var contract = CONTRACT_ADDRESS;
10 var seller = SELLER_ADDRESS;
11 var buyer = BUYER_ADDRESS;
12 var state = NULL_STATE;
13 var value = INITIAL_VALUE;
14 #alphabet Constructor {smart_contract_initialized};
15 Constructor() =
16   constructor?msg_sender.msg_value ->
17   ifa (msg_value % 2 == 0) {
18     seller_set_to.msg_sender{seller = msg_sender;}
19     value_as_item_price_set_to_half_of_guaranty.
20     msg_value{value = msg_value / 2;} ->
21     state_transitioned_to_created{state =
22     CREATED_STATE;} ->
23     smart_contract_initialized -> Purchase()
24   } else { odd_msg_value.exception.msg_value ->
25   Constructor(); }
26 #alphabet Abort {aborted, end};
27 Abort() =
28   abort?msg_sender.msg_value ->
29   ifa (msg_sender == seller) {
30     ifa (state == CREATED_STATE) {
31       state_transitioned_to_inactive{state =
32       INACTIVE_STATE;} ->
33       contract_balance_transferred_to_seller.
34       balances[contract].seller{
35         balances[seller] = balances[seller] +
36         balances[contract];
37         balances[contract] = 0; } ->
38       aborted -> end -> Skip
39     } else { non_created_state_exception.state ->
40     end -> Skip }
41   } else { non_seller_exception.msg_sender ->
42   end -> Skip };
43
44 #alphabet ConfirmPurchased {purchase_confirmed, end};
45 ConfirmPurchased() =
46   confirm_purchased?msg_sender.msg_value ->
47   ifa (state == CREATED_STATE) {
48     ifa (msg_value == 2 * value) {
49       state_transitioned_to_locked{state = LOCKED_STATE
50       }; ->
51       contract_balance_added_with_deposit_of_msg_value{
52       balances[contract] = balances[contract] +
53       msg_value;
54       if (msg_sender == BUYER_ADDRESS) {
55         balances[buyer] = balances[buyer] - msg_value;
56       } else { balances[seller] = balances[seller] -
57       msg_value; }
58       } -> buyer_set_to.msg_sender{buyer = msg_sender;}
59       } -> purchase_confirmed -> end -> Skip
60     } else { non_matching_msg_value.exception.msg_value
61     -> end -> Skip }
62   } else { non_created_state_exception.state -> end ->
63   Skip };
64 #alphabet ConfirmReceived {item_received, end};
65 ConfirmReceived() =
66   confirm_received?msg_sender.msg_value ->
67   ifa (msg_sender == buyer) {
68     if (state == LOCKED_STATE) {
69       state_transitioned_to_release{state =
70       RELEASE_STATE;} ->
71       half_of_deposit_returned_to_buyer.value.msg_sender
72       {
73         balances[contract] = balances[contract] - value;
74         if (msg_sender == BUYER_ADDRESS) {
75           balances[buyer] = balances[buyer] + value;
76         } else {
77           balances[seller] = balances[seller] + value
78           ; } -> contract_balance_transferred_to_seller.
79           balances[contract].seller{ balances[seller] =
80           balances[seller] + balances[contract];
81           balances[contract] = 0;
82           } -> item_received -> end -> Skip
83         } else { non_locked_state_exception.state ->
84         end -> Skip }
85       } else { non_buyer_exception.msg_sender -> end ->
86       Skip };

```

Fig. 5. Correct Model of Safe Remote Purchase (part 1).

for other concerns, including a faithful translation of the possibility of never-ending invocations of functions exposed by an alive contract on the chain and the incorporation of the environmental factors such as consensus clients into the framework to leave space for further extensions as part of the future work.

5.1 Correct CSP# Modelling

Following the approach, in the case study, the business logic model is roughly the same as the previous smart contract modelling in Fig. 4, except that tiny modifications (detailed later) and syntactical renaming of events (see Fig. 5). In addition, we added recursion to model that a function can be called multiple times, line 1–3 in Fig. 6.

The newly added model component is the control logic and its interface shown in Fig. 7. In the control logic, we model an abstracted blockchain (line 34) consisting of Blockchain nodes (line 33) running a consensus algorithm (line 27). Each blockchain node has an EVM that stores all the smart contract functions and executes functions to update the global state upon invocation. In the case study, this is modelled by allowing the node to be able to execute all the smart contract functions *i.e.*, calling the interface of the smart contracts (line 24). Since the EVM is a single machine, meaning that each function is executed atomically, the relations between the executions are internal choices capturing that once called, the function execution is not interrupted by other functions in the same EVM, which is

the key to avoiding false attacks. The non-determinism is modelled by the uncertainty of the node who proposes the block, *i.e.*, which node is the next to execute the triggered smart contract functions in EVM. In reality, the consensus algorithm decides the block proposer. There are various consensus algorithms and modelling them in details is challenging as stated in [22]. In this work, serving the purpose of reducing false attacks, does not require full details of the consensus algorithms. Therefore, we model an abstracted version in the consensusClient process (line 22–26), where the nodes take turns to be the block proposer. While some blockchain uses this model (*e.g.*, Tendermint implements a round-robin approach to decide the schedule of nodes), it is indeed a naive approach that may suffer from attacks (*e.g.*, malicious nodes may prepare a fork of the chain to launch double spending knowing the scheduling of the proposers). We assume the nodes are honest and leave the full detail modelling of consensus to future work.

The left part of Fig. 7 (line 10–21) models the interfaces between the actual smart contract model (business logic) and the blockchain model (control logic). That is, once a proposer is decided in the ConsensusClient, the corresponding node executes ExecutionClient to call the smart contract function in the node’s transaction pool. We assume there is only one transaction in a block to ensure the occurrence of *Multi-Transactional* behaviour. Extending to multiple transaction executions is easy by making them recursive. The calling of the transaction is implemented in the corresponding interface. Once called, the interface sends a message to trigger the actual smart contract function defined in the smart contract model (the business logic in Fig. 5), thus linking the control logic (the blockchain) with the business logic.

Note that the events are renamed to provide meaningful information, compared to the replicated model where the same event names are used as in the [16], but they are semantically equivalent. In addition, to enable the interface to work correctly, the process of each smart contract function added a receiving message event at the beginning to model receiving signals from the interface *i.e.*, line 14, 23, 37 and 51 in Fig. 5.

```

1  AbortRC() = Abort(); AbortRC();
2  ConfirmPurchasedRC() = ConfirmPurchased();
   ConfirmPurchasedRC();
3  ConfirmReceivedRC() = ConfirmReceived();
   ConfirmReceivedRC();
4  Purchase() = AbortRC() ||| ConfirmPurchasedRC() |||
   ConfirmReceivedRC();
5  Deployer(address, guaranty_ether_value) =
6  constructor(address.guaranty_ether_value -> Skip;
7  Seller(address) =
8  abort_invocation!address.0 -> Seller(address);
9  Buyer(address, deposit_ether_value) =
10 confirm_purchased_invocation!address.
   deposit_ether_value ->
11 Buyer(address, deposit_ether_value)
12 [] confirm_received_invocation!address.0 ->
13 Buyer(address, deposit_ether_value);
14 BlockchainSystem() = Constructor() |||
15 (Deployer(SELLER_ADDRESS, ITEM_PRICE_VALUE * 2);
16 ((Seller(SELLER_ADDRESS) ||| Buyer()) |||
17 (Buyer(BUYER_ADDRESS, ITEM_PRICE_VALUE * 2)
18 ||| Seller(BUYER_ADDRESS)))
19 ) ||| BlockchainNetwork();
20 #define buyer_balance_remains_unchanged (balances[buyer] =
   = INITIAL_BALANCE);
21 #define seller_balance_remains_unchanged (balances[seller]
   = INITIAL_BALANCE);
22 #define buyer_balance_deducted_by_item_price_value (
   balances[buyer] == INITIAL_BALANCE -
   ITEM_PRICE_VALUE);
23 #define seller_balance_added_by_item_price_value (balances
   [seller] == INITIAL_BALANCE + ITEM_PRICE_VALUE);
24 #define buyer_is_BUYER_ADDRESS (buyer == BUYER_ADDRESS);
25 #define buyer_is_SELLER_ADDRESS (buyer == SELLER_ADDRESS);
26 #assert BlockchainSystem() [= []] (aborted -> << (
   buyer_balance_remains_unchanged &&
   seller_balance_remains_unchanged));
27 #assert BlockchainSystem() [= []] (item_received &&
   buyer_is_BUYER_ADDRESS -> << (
   buyer_balance_deducted_by_item_price_value &&
   seller_balance_added_by_item_price_value));
28 #assert BlockchainSystem() [= []] (item_received &&
   buyer_is_SELLER_ADDRESS -> << (
   buyer_balance_remains_unchanged &&
   seller_balance_remains_unchanged));
29 #assert BlockchainSystem() reaches non_constant_balances;
    
```

Fig. 6. Correct Model of Safe Remote Purchase (part 2).

```

1  #define N 3; #define INITIAL_BALANCE 100;
2  #define EXIT_CODE_SUCCESS 0; #define EXIT_CODE_ERROR 1;
3  var balances = [0, INITIAL_BALANCE, INITIAL_BALANCE];
4  hvar counter = 0;
5  channel abort_invocation 0; channel abort 0;
6  channel confirm_purchased_invocation 0;
7  channel confirm_purchased 0;
8  channel confirm_received_invocation 0;
9  channel confirm_received 0; channel release 0;
10 AbortExecutor() =
11   abort_invocation?msg_sender.msg_value ->
12   abort!msg_sender.msg_value ->
13   release?exit_code -> Skip;
14 ConfirmPurchasedExecutor() =
15   confirm_purchased_invocation?msg_sender.msg_value ->
16   confirm_purchased!msg_sender.msg_value ->
17   release?exit_code -> Skip;
18 ConfirmReceivedExecutor() =
19   confirm_received_invocation?msg_sender.msg_value ->
20   confirm_received!msg_sender.msg_value ->
21   release?exit_code -> Skip;
22 ExecutionClient(i) =
23   (start_execution_client.i -> Skip);
24   (AbortExecutor() [] ConfirmPurchasedExecutor() []
25    ConfirmReceivedExecutor());
25   (end_execution_client.i -> Skip);
26 ExecutionClient(i);
27 ConsensusClient(i) =
28   [counter == i]
29   start_execution_client.i ->
30   end_execution_client.i ->
31   tau{counter = (counter + 1) % N} ->
32   ConsensusClient(i);
33 BlockchainNode(i) = ExecutionClient(i) ||
34   ConsensusClient(i);
35 BlockchainNetwork() = BlockchainNode(0) ||
36   BlockchainNode(1) || BlockchainNode(2);
37 #define non_constant_balances (balances[0] +
38   balances[1] + balances[2] != 0 +
39   INITIAL_BALANCE + INITIAL_BALANCE);

```

Fig. 7. Modelling the Blockchain in Control Logic (right) and its Interface (left).

Verification Results. The verification results produced by running our CSP# model in the PAT model checker are shown in Table 2. Given that the assertions inherently convey their semantics, we refrain from reiterating them for the sake of brevity. The results prove *non-existence* of any vulnerabilities pertaining to the interleaved executions of contract functions and *infeasibility* of launching the attack described in Sect. 4.2 in practice to steal the funds deposited by the buyer, via demonstrating the validity of the last three assertions and the invalidity of the first assertion contradictory to the conclusion drawn from Table 1. We therefore conclude that the original contract design is in fact robust to any finely crafted attacks in the form of well planned sequences of function invocations and the result reported in the original literature [16] is challenged and shown to be a *false positive* alert based on our model built under the correct assumption of the execution model of Ethereum blockchain.

Modelling Strategy Comparisons. Essentially, integrating the blockchain framework into the modelling process consistently enforces mutual exclusive access to the execution of each constituent process that represents its respective contract function. This is achieved through cooperatively running execution clients and consensus clients of all participating nodes, as represented by the pro-

Table 2. Verification Result of Our Correct CSP# Model.

Assertion	Validity
BlockchainSystem() reaches non_constant_balances	NOT VALID
BlockchainSystem() $\text{---} \square[(\text{item_received} \ \&\& \ \text{buyer_is_SELLER_ADDRESS} \ -i \ j_i(\text{buyer_balance_remains_unchanged} \ \&\& \ \text{seller_balance_remains_unchanged}))$	VALID
BlockchainSystem() $\text{---} \square[(\text{item_received} \ \&\& \ \text{buyer_is_BUYER_ADDRESS} \ -i \ j_i(\text{buyer_balance_deducted_by_item_price_value} \ \&\& \ \text{seller_balance_added_by_item_price_value}))$	VALID
BlockchainSystem() $\text{---} \square[(\text{aborted} \ -i \ j_i(\text{buyer_balance_remains_unchanged} \ \&\& \ \text{seller_balance_remains_unchanged}))$	VALID

cess `BlockchainNetwork` in our framework. Despite the semantic implications of the actual construct used—either interleaving operators or general choice operators—in defining the process, this integration characterises all possible interactions with the contract.

We have also effectively tackled the oversight of the potential scenario of successive attempts to invoke the function `Abort`, `ConfirmPurchased`, and `ConfirmReceived` multiple times in unanticipated order, independent of the current state of the contract (*e.g.*, even after the finalisation of the purchase order), so as to expand the derived state space to encompass these non-terminating behaviours, ensuring comprehensive coverage of their side effects towards the overall correctness of the contract. In addition, to ensure it works correctly, their soundness criteria must also be properly model-checked during the verification phase, leading to the extra assertions and verification results in Table 2.

Correctness. The correctness of this CSP# model mostly lies in the accuracy of our framework with respect to the correspondence between the actual execution model of Ethereum and constructs used for modelling their abstracted behaviours (*e.g.*, scheduling of smart contract function executions), in particular, the use of those synchronous channels for both inter-node and intra-node communications, the use of the general choice operator in defining the process of the execution client, as well as the way in which a simplified blockchain node is defined in terms of cooperatively running execution/consensus clients. We have provided detailed justification in our Github report; for the sake of brevity, no further explanations are given here.

Limitations. In the current model, intricacies of network data transmission have been fully abstracted away from our framework and replaced by reliable channels for the sake of simplicity, and all possible interactions initiated by a third-party proxy contract have been intentionally omitted due to possible combinations of such interactions being infinite. We modelled an ideal consensus omitting failure in consensus mechanism *e.g.*, the actual order in which transactions are processed may differ from the expected order of executions when assuming no failures during the consensus procedure if dismissing this aspect of the environment when model checking any smart contract of interest.

5.2 A Generalized Formal Framework in CSP#

To generalise the approach beyond the Safe Remote Purchase case study, we formalised it into a framework. In particular, we implemented a script (see Fig. 8) to generate the control logic automatically. Since the “business logic” of a smart contract is application-specific, it needs to be modelled manually.

To test the model generator and evaluate that the approach would not miss out on concurrency attacks, we verified the smart contract in Example 1 that truly suffers from concurrency issues to confirm the ability of our approach to detect concurrency vulnerabilities. We manually modelled the business logic of

```

1  const fs = require("fs");
2  fs.readFile("input.json", "utf8", (err, data) => {
3    if (err) {console.error(err); return;}
4    try {
5      const capitalize = (string) => string.charAt(0)
6        .toLowerCase().toUpperCase() + string.slice(1);
7      const jsonData = JSON.parse(data);
8      const outputString =
9        "#define N " + jsonData.n + ";\n" +
10       "#define INITIAL_BALANCE " + jsonData.
11         initialBalance + ";\n" +
12       "#define EXIT_CODE_SUCCESS 0;\n" +
13       "#define EXIT_CODE_ERROR 1;\n" +
14       "var balances = [0, " + (new Array(jsonData.
15         .n - 1).fill("INITIAL_BALANCE")).join(
16         ", ") + "];\n" +
17       "hvar counter = 0;\n" +
18       jsonData.functionNames.map((functionName)
19         => "channel " + functionName + "
20         _invocation 0;\n" +
21         "channel " + functionName + " 0;\n"
22         ).join('') +
23       "channel release 0;\n" +
24       jsonData.functionNames.map((functionName)
25         =>
26         functionName.split('_').map(capitalize).
27         join('') + "Executor() = \n" +
28         " " + functionName + "_invocation?
29         msg_sender.msg_value -> \n" +
30         " " + functionName + "msg_sender.
31         msg_value -> \n" +
32         " " + "release?exit_code -> \n" + " Skip;\n"
33         ).join('\n') + '\n' +
34
35       "ExecutionClient(i) = \n" +
36       " (start_execution_client.i -> Skip);\n" +
37       " (" + jsonData.functionNames.map((functionName)
38         => functionName.split('_').map(capitalize)
39         .join('') + "Executor()").join(" [] ") + ")
40       ;\n" +
41       " (end_execution_client.i -> Skip);\n" +
42       " ExecutionClient(i);\n" +
43       "ConsensusClient(i) = \n" +
44       " [counter == i]\n" +
45       " start_execution_client.i -> \n" +
46       " end_execution_client.i -> \n" +
47       " tau{counter = (counter + 1) % N} -> \n" +
48       " ConsensusClient(i);\n" +
49
50       "BlockchainNode(i) = ExecutionClient(i) ||
51       ConsensusClient(i);\n" +
52       "BlockchainNetwork() = BlockchainNode(0) ||
53       BlockchainNode(1) || BlockchainNode(2);\n" +
54       "\n" +
55       "#define non_constant_balances (" + Array.from(
56       Array(jsonData.n).keys()).map((index) => "
57       balances[" + index + "]" + ").join(' ') + "
58       != 0 + " + (new Array(jsonData.n - 1).fill(
59       "INITIAL_BALANCE")).join(" + ") + ");\n";
60       fs.writeFile("./blockchain_framework.csp",
61       outputString, (err) => {
62         if (err) { console.error(err); return; }
63         console.log("The blockchain framework to be
64         included in your CSP model has been
65         successfully generated under the same
66         directory of this generator script.");});
67     } catch (parseError) {
68       console.error(parseError); } });

```

Fig. 8. Blockchain Model Generator.

```

1  #define N 3; #define INITIAL_BALANCE 100;
2  #define EXIT_CODE_SUCCESS 0;
3  #define EXIT_CODE_ERROR 1;
4  var balances = [0, INITIAL_BALANCE,
5    INITIAL_BALANCE];
6  hvar counter = 0;
7  channel get_invocation 0; channel get 0;
8  channel set_invocation 0;
9  channel set 0; channel release 0;
10 GetExecutor() =
11   get_invocation?msg_sender.msg_value ->
12   get?msg_sender.msg_value ->
13   release?exit_code -> Skip;
14 SetExecutor() =
15   set_invocation?msg_sender.msg_value ->
16   set?msg_sender.msg_value ->
17   release?exit_code -> Skip;
18
19 ExecutionClient(i) =
20   (start_execution_client.i -> Skip);
21   (GetExecutor() [] SetExecutor());
22   (end_execution_client.i -> Skip); ExecutionClient(i);
23 ConsensusClient(i) =
24   [counter == i] start_execution_client.i ->
25   end_execution_client.i ->
26   tau{counter = (counter + 1) % N} ->
27   ConsensusClient(i);
28 BlockchainNode(i) = ExecutionClient(i) ||
29   ConsensusClient(i);
30 BlockchainNetwork() = BlockchainNode(0) ||
31   BlockchainNode(1) || BlockchainNode(2);
32 #define non_constant_balances (balances[0] + balances[1]
33   + balances[2] != 0 + INITIAL_BALANCE +
34   INITIAL_BALANCE);

```

Fig. 9. CSP# Blockchain Framework for Example 1 Smart Contract.

the smart contract¹. The generated model of the “control logic” smart contract is shown in Fig. 9, where the main difference lies in the generated interface (line 9-16) and their invocation (line 19) in the process of the blockchain nodes *i.e.*, ExecutionClient. We then integrated the smart contract model with the generated model. The verification result confirms the expected concurrency vulnerability.

These two case studies demonstrate that our framework empowers users to model with greater precision and accuracy, the *non-determinism* stemming from transaction races *i.e.*, invocation requests to contract functions, which are often mishandled by average model engineers, leading to the inclusion of unlikely execution sequences in the state space for verification, resulting in false attack alarm and unnecessary labour for rectifying them. Our primary contribution lies in reducing *false positive* cases by correctly modelling all possible transac-

¹ The modelling is straightforward and thus we do not explain them here. For details, refer to our Github.

tion orders and reducing *false negative* cases by including potential conflicting transaction executions in the state space. Leveraging our generator script, which separates *control logic* from *business logic*, enables more diverse modelling strategies: Users can use the interleaving operator to compose the smart contract functions without considering how exactly the underlying blockchain schedules the actual execution sequence; The common application-agnostic details observed in all types of smart contract are abstracted into a separate model component *i.e.*, the blockchain framework, for automatic generation. This framework frees the model engineers from the need to possess this knowledge, thereby enhancing the *generalizability* of our framework.

6 Related Works

Security of smart contracts and blockchain vulnerabilities have been extensively surveyed in [4, 9, 13]. Concurrency issues discussed in this work spans over several surveyed attacks, as shown in Sect. 2.

To address smart contract vulnerabilities, various verification techniques (surveyed in [2]) and tools (surveyed in [3]) have been developed, including testing-based approaches (like [11]) and static analysis based approaches (*e.g.*, symbolic execution [13]). These approaches heavily rely on known patterns and cannot guarantee correctness and security. On the other hand, formal verification approaches that overcome such limitations, have also been developed (surveyed in [24]) including theorem proving based approaches *e.g.*, [20], model checking approaches *e.g.*, [12] and abstract state machines based approaches *e.g.*, [5, 6]. However, these approaches verify smart contracts independently. Exceptions that consider other participants in specific attacks have been discussed in Sect. 1. In contrast, there are much less works on formal verification of blockchain due to its complexity, with exceptions including [7, 14, 23].

The concurrency issue in this work involves the specification and verification of both smart contracts and their blockchain environment. Notable techniques for addressing concurrency issues have been introduced in Sect. 1.

7 Conclusions and Future Work

Aiming to assist model engineers in constructing correct and precise models of smart contracts that involve *non-determinism* arising from the races between transactions, which are often overlooked or mistakenly handled by an average model engineer, this work proposes a formal framework that facilitates a holistic inclusion of transaction executions to reduce the false attacks in verifying concurrency of smart contracts, achieved by recognising the commonality of the separation of the control logic from the business logic of the smart contracts.

Recognising the limitations of the framework (as mentioned earlier), the primary focus of future work is to address them by offering more detailed consensus and network integration. Additionally, another area for future exploration involves integrating the other types of concurrency in smart contract execution into the framework, where the challenge is the necessary manual translation.

References

1. Post-mortem investigation (2016). <https://www.kingoftheether.com/postmortem.html>. Accessed 11 Apr 2024
2. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: a survey. *Pervasive Mob. Comput.* **67**, 101227 (2020)
3. Angelo, M.D., Salzer, G.: A survey of tools for analyzing ethereum smart contracts. In: DAPPS, pp. 69–78 (2019)
4. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: POST, pp. 164–186 (2017)
5. Braghin, C., Riccobene, E., Valentini, S.: An asm-based approach for security assessment of ethereum smart contracts. In: SECRYPT, pp. 334–344 (2024)
6. Braghin, C., Riccobene, E., Valentini, S.: Modeling and verification of smart contracts with abstract state machines. In: SAC, pp. 1425–1432 (2024)
7. Braithwaite, S., et al.: Formal specification and model checking of the tendermint blockchain synchronization protocol (short paper). In: FMBC (2020)
8. Breidenbach, L., Juels, P.D.A., Siringu, E.G.: An in-depth look at the parity multisig bug (2017). <https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/>. Accessed 11 Apr 2024
9. Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on ethereum systems security: vulnerabilities, attacks, and defenses. *ACM Comput. Surv.* **53**(3) (2021)
10. Daian, P., et al.: Flash boys 2.0: frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: SP, pp. 910–927 (2020)
11. Grieco, G., Song, W., Cygan, A., Feist, J., Groce, A.: Echidna: effective, usable, and fast fuzzing for smart contracts. In: ISSTA, pp. 557–560 (2020)
12. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: analyzing safety of smart contracts. In: NDSS, pp. 1–12 (2018)
13. Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: CCS, pp. 254–269 (2016)
14. Maung Maung Thin, W.Y., Dong, N., Bai, G., Dong, J.S.: Formal analysis of a proof-of-stake blockchain. In: ICECCS, pp. 197–200 (2018)
15. Munir, S., Reichenbach, C.: Todler: a transaction ordering dependency analyzer - for ethereum smart contracts. In: WETSEB, pp. 9–16 (2023)
16. Qu, M., Huang, X., Chen, X., Wang, Y., Ma, X., Liu, D.: Formal verification of smart contracts from the perspective of concurrency. In: Qiu, M. (ed.) *SmartBlock 2018*. LNCS, vol. 11373, pp. 32–43. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-05764-0_4
17. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) *FC 2017*. LNCS, vol. 10323, pp. 478–493. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_30
18. Siegel, D.: Understanding the DAO attack. <https://www.coindesk.com/learn/understanding-the-dao-attack/>. Accessed 11 Apr 2024
19. Solidity: Solidity by example safe remote purchase. <https://docs.soliditylang.org/en/latest/solidity-by-example.html>. Accessed 16 Apr 2024
20. Sotnichek, M.: Formal verification of smart contracts with the k framework (2019). <https://www.apriorit.com/dev-blog/592-formal-verification-with-k-framework>. Accessed 23 Sept 2024
21. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) *CAV 2009*. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_59

22. Thin, W.Y.M.M., Dong, N., Bai, G., Dong, J.S.: Formal analysis of a proof-of-stake blockchain. In: ICECCS, pp. 197–200 (2018)
23. Tholoniati, P., Gramoli, V.: Formal verification of blockchain byzantine fault tolerance. In: Handbook on Blockchain, pp. 389–412 (2022)
24. Tolmach, P., Li, Y., Lin, S., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. *ACM Comput. Surv.* **54**(7), 148:1–148:38 (2022)
25. Wang, Y., Li, J., Liu, W., Tan, A., Derhab, A.: Efficient concurrent execution of smart contracts in blockchain sharding. *Secur. Commun. Netw.* (2021)
26. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger (2019). <https://ethereum.github.io/yellowpaper/paper.pdf>
27. Xu, X., Pautasso, C., Zhu, L., Lu, Q., Weber, I.: A pattern collection for blockchain-based applications. In: EuroPLoP, pp. 3:1–3:20 (2018)

Author Index

A

Aït Aneur, Yamine 314
Aksoy, Kubra 162
An, Dongdong 146
André, Étienne 37, 51

B

Bonnah, Ernest 332
Brechelmacher, Otto 216
Brunel, Julien 375
Bultan, Tevfik 125

C

Chen, Si 257

D

DeLong, Erin 125
Deniz, Elif 297
Dépernet, Sarah 51
Dima, Catalin 181
Dong, Naipeng 391
Doose, David 375
Dou, Guowei 257
Downing, Mara 125
Dupont, Guillaume 314

E

Eiers, William 125

F

Fu, Zhiyuan 237

G

Glesner, Sabine 85

H

Hammami, Mariem 181
Hoque, Khaza Anuarul 332
Hou, Zhe 391
Huang, Yanhong 1, 146

I

Ishikawa, Fuyuki 18, 314

J

Jiang, Jiacheng 237

K

Kadron, Ismet Burak 125
Klein, Julian 85
Kobayashi, Tsutomu 18, 314
Kogel, Paul 85

L

Laleau, Régine 181
Lefaucheux, Engel 51
Lin, Chen-Kai 199
Liu, Jing 146
Liu, Shaoying 106
Liu, Xu 146
Liu, Zhiming 278
Lodha, Anushka 125

M

Minh Do, Canh 353

N

Nguyen, Luan Viet 332
Ničković, Dejan 216
Nießen, Tobias 216

O

Ogata, Kazuhiro 353
Oualhadj, Youssouf 181
Ozawa Burns, Brian 125

P

Pang, Jun 278

Q

Qin, Shengchao 1, 146, 237

R

Rashid, Adnan 162

Rivière, Peter 314

S

Sallinger, Sarah 216

Shang, Yuxiang 106

Shi, Jianqi 1, 146

Singh, Neeraj Kumar 314

Song Dong, Jin 391

Sproston, Jeremy 70

T

Tahar, Sofène 162, 297

Takagi, Tsubasa 353

Tao, Ran 1

W

Wang, Bow-Yaw 199

Weissenbacher, Georg 216

Wen, Cheng 237

Wu, Guisen 278

X

Xu, Zhiwu 237

Y

Yang, Yang 1, 146

Yu, Wensheng 257

Yu, Yisong 391

Z

Zhang, Hao 146

Zhang, Ru 257

Zhao, Mengyan 1

Zhao, Qin 146