Xinyu Feng · Markus Müller-Olm
Zijiang Yang (Eds.)

LNCS 10998

Dependable Software Engineering

Theories, Tools, and Applications

4th International Symposium, SETTA 2018
Beijing, China, September 4–6, 2018
Proceedings

Springer

More information about this series at http://www.springer.com/series/7408

Xinyu Feng · Markus Müller-Olm
Zijiang Yang (Eds.)

Dependable Software Engineering

Theories, Tools, and Applications

4th International Symposium, SETTA 2018
Beijing, China, September 4–6, 2018
Proceedings

Springer

Editors
Xinyu Feng
Nanjing University
Nanjing
China

Zijiang Yang
Western Michigan University
Kalamazoo, MI
USA

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster
Münster
Germany

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-99932-6 ISBN 978-3-319-99933-3 (eBook)
https://doi.org/10.1007/978-3-319-99933-3

Library of Congress Control Number: 2018953821

LNCS Sublibrary: SL2 – Programming and Software Engineering

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This volume contains the papers presented at the 4th International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA 2018), held during September 4–6, 2018, in Beijing. The purpose of SETTA is to provide an international forum for researchers and practitioners to share cutting-edge advancements and strengthen collaborations in the field of formal methods and its interoperability with software engineering for building reliable, safe, secure, and smart systems. Past SETTA symposiums were successfully held in Nanjing (2015), Beijing (2016), and Changsha (2017).

SETTA 2018 solicited submissions in two categories, regular papers and short papers. Short papers could discuss ongoing research at an early stage, or present systems and tools. There were 22 submissions in total. Each submission was reviewed by at least three, and on average 3.7, Program Committee (PC) members, with the help of external reviewers. After thoroughly evaluating the relevance and quality of each paper through online PC meetings, the PC decided to accept nine regular papers and three short papers. The program also included three invited talks given by Prof. Moshe Vardi from Rice University, Prof. Tao Xie from University of Illinois at Urbana-Champaign, and Prof. Hongseok Yang from KAIST. Prof. Moshe Vardi was a joint keynote speaker of CONFESTA 2018, a joint event comprising the international 2018 conferences CONCUR, FORMATS, QEST, and SETTA, alongside with several workshops and tutorials.

This program would not have been possible without the unstinting efforts of many people, whom we would like to thank. First, we would like to express our gratitude to the PC and the external reviewers for their hard work put in toward ensuring the high quality of the proceedings. Our thanks also go to the Steering Committee for its advice and help. We would like to warmly thank the general chair of SETTA 2018, Prof. Chaochen Zhou, the general chair of CONFESTA 2018, Prof. Huimin Lin, the local organizers including Dr. David N. Jansen, Dr. Andrea Turrini, Dr. Shuling Wang, Dr. Peng Wu, Dr. Zhilin Wu, Dr. Bai Xue, Prof. Lijun Zhang, and all others on the local Organizing Committee.

We also enjoyed great institutional and financial support from the Institute of Software, Chinese Academy of Sciences, without which an international conference like CONFESTA and the co-located events could not have been successfully organized. We also thank the Chinese Academy of Sciences and the other sponsors for their financial support. Furthermore, we would like to thank Springer for sponsoring the Best Paper Award. Finally, we are grateful to the developers of the EasyChair system, which significantly eased the processes of submission, paper selection, and proceedings compilation.

July 2018

Xinyu Feng
Markus Müller-Olm
Zijiang Yang

Organization

General Chair

Chaochen Zhou	Institute of Software, CAS, China

Program Chairs

Xinyu Feng	Nanjing University, China
Markus Müller-Olm	Westfälische Wilhelms-Universität Münster, Germany
Zijiang Yang	Western Michigan University, USA

Program Committee

Farhad Arbab	CWI and Leiden University, The Netherlands
Sanjoy Baruah	Washington University in St. Louis, USA
Lei Bu	Nanjing University, China
Michael Butler	University of Southampton, UK
Yan Cai	Institute of Software, CAS, China
Taolue Chen	Birkbeck, University of London, UK
Yuxin Deng	East China Normal University, China
Xinyu Feng	Nanjing University, China
Yuan Feng	University of Technology, Sydney, Australia
Ernst Moritz Hahn	Institute of Software, CAS, China
Dan Hao	Peking University, China
Maritta Heisel	University of Duisburg-Essen, Germany
Raymond Hu	Imperial College London, UK
He Jiang	Dalian University of Technology, China
Yu Jiang	Tsinghua University, China
Einar Broch Johnsen	University of Oslo, Norway
Guoqiang Li	Shanghai Jiao Tong University, China
Ting Liu	Xi'an Jiaotong University, China
Tongping Liu	University of Texas at San Antonio, USA
Yang Liu	Nanyang Technological University, Singapore
Xiapu Luo	The Hong Kong Polytechnic University, Hong Kong, SAR China
Stephan Merz	Inria Nancy and LORIA, France
Markus Müller-Olm	Westfälische Wilhelms-Universität Münster, Germany
Jun Pang	University of Luxembourg, Luxembourg
Davide Sangiorgi	University of Bologna, Italy
Oleg Sokolsky	University of Pennsylvania, USA
Fu Song	ShanghaiTech University, China
Zhendong Su	University of California, Davis, USA

Jun Sun	Singapore University of Technology and Design, Singapore
Walid Mohamed Taha	Halmstad University and University of Houston, Sweden
Sofiene Tahar	Concordia University, Canada
Cong Tian	Xidian University, China
Bow-Yaw Wang	Academia Sinica, Taiwan
Chao Wang	University of Southern California, USA
Ji Wang	National University of Defense Technology, China
Heike Wehrheim	University of Paderborn, Germany
Xin Xia	Monash University, Australia
Zijiang Yang	Western Michigan University, USA
Shin Yoo	KAIST, Korea

Local Organizing Committee

Jansen, David N.
Lv, Yi
Turrini, Andrea
Wang, Shuling
Wu, Peng
Wu, Zhilin (chair)
Xue, Bai
Yan, Rongjie
Zhu, Xueyang

Additional Reviewers

Chen, Liqian
Chen, Zhe
Colley, John
Dongol, Brijesh
Elderhalli, Yassmeen
Gu, Tianxiao
Gutsfeld, Jens
Kaur, Ramneet
Kenter, Sebastian
Kharraz, Karam
König, Jürgen
Liu, Hongyu
Petre, Luigia
Santen, Thomas
Silvestro, Sam
Soualhia, Mbarka
Steffen, Martin
Tang, Enyi
Zhang, Min
Zhu, Chenyang

Abstracts of Invited Talks

Intelligent Software Engineering: Synergy between AI and Software Engineering

Tao Xie

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
taoxie@illinois.edu

Abstract. As an example of exploiting the synergy between AI and software engineering, the field of intelligent software engineering has emerged with various advances in recent years. Such field broadly addresses issues on *intelligent* [software engineering] and [*intelligence software*] engineering. The former, *intelligent* [software engineering], focuses on instilling intelligence in approaches developed to address various software engineering tasks to accomplish high effectiveness and efficiency. The latter, [*intelligence software*] engineering, focuses on addressing various software engineering tasks for intelligence software, e.g., AI software. In this paper, we discuss recent research and future directions in the field of intelligent software engineering.

This work was supported in part by National Science Foundation under grants no. CNS-1513939 and CNS1564274, and a grant from the ZJUI Research Program.

Formal Semantics of Probabilistic Programming Languages: Issues, Results and Opportunities

Hongseok Yang

KAIST, Daejeon, South Korea
hongseok00@gmail.com

Probabilistic programming refers to the idea of developing a programming language for writing and reasoning about probabilistic models from machine learning and statistics. Such a language comes with the implementation of several generic inference algorithms that answer various queries about the models written in the language, such as posterior inference and marginalisation. By providing these algorithms, a probabilistic programming language enables data scientists to focus on designing good models based on their domain knowledge, instead of building effective inference engines for their models, a task that typically requires expertise in machine learning, statistics and systems. Even experts in machine learning and statistics may get benefited from such a probabilistic programming system because using the system they can easily explore highly advanced models.

In the past three years, I and my colleagues have worked on developing so called denotational semantics of such probabilistic programming languages, especially those that support expressive language features such as higher-order functions, continuous distributions and general recursion. Such semantics describe what probabilistic model each program in those languages denotes, serve as specifications for inference algorithms for the languages, and justify compiler optimisations for probabilistic programs or models. In this talk, I will describe what we have learnt so far, and explain how these lessons help improve the design and implementation of these probabilistic programming languages and their inference engines.

Contents

Abstracts of Invited Talks

Intelligent Software Engineering: Synergy Between AI and Software Engineering

Tao Xie(✉)

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
taoxie@illinois.edu

Abstract. As an example of exploiting the synergy between AI and software engineering, the field of intelligent software engineering has emerged with various advances in recent years. Such field broadly addresses issues on *intelligent* [software engineering] and [*intelligence software*] engineering. The former, *intelligent* [software engineering], focuses on instilling intelligence in approaches developed to address various software engineering tasks to accomplish high effectiveness and efficiency. The latter, [*intelligence software*] engineering, focuses on addressing various software engineering tasks for intelligence software, e.g., AI software. In this paper, we discuss recent research and future directions in the field of intelligent software engineering.

Keyword: Intelligent software engineering

1 Introduction

As an example of exploiting the synergy between AI and software engineering, the field of intelligent software engineering [31] has emerged with various advances in recent years. Such field broadly addresses issues on *intelligent* [software engineering] and [*intelligence software*] engineering. The former, *intelligent* [software engineering], focuses on instilling intelligence in approaches developed to address various software engineering tasks to accomplish high effectiveness and efficiency. The latter, [*intelligence software*] engineering, focuses on addressing various software engineering tasks for intelligence software, e.g., AI software. Indeed, the preceding two aspects can overlap when instilling intelligence in approaches developed to address various software engineering tasks for intelligence software. By nature, the field of intelligent software engineering is a research field spanning at least the research communities of software engineering and AI.

This work was supported in part by National Science Foundation under grants no. CNS-1513939 and CNS1564274, and a grant from the ZJUI Research Program.

2 Instilling Intelligence in Software Engineering

Applying or adapting AI technologies to address various software engineering tasks [13] has been actively pursued by researchers from the software engineering research community, e.g., machine learning for software engineering [1,18,38] and natural language processing for software engineering [20,21,29,30,39,40], and also by researchers from the AI research community in recent years [2] partly due to the increasing popularity of deep learning [24]. Much of such previous work has been on automating as much as possible to address a specific software engineering task such as programming and testing. But as pointed out by various AI researchers [14,17], AI technologies typically enhance or augment human, instead of replacing human.

In future work, we envision that intelligence can be instilled into approaches for software engineering tasks in the following two example ways as starting points.

Natural Language Interfacing. Natural language conversations between a human and a machine can be traced back to the Turing test [28], proposed by Turing in 1950, as a test for a machine to exhibit intelligent behaviors indistinguishable from a human's. Natural-language-based chatbots have been increasingly developed and deployed for various domains: virtual assistants (such as Apple Siri, Google Assistant, Amazon Alexa, Microsoft Cortana, Samsung Bixby), customer services, social media (such as Facebook Messenger chatbots). Very recently, exploring the use of chatbots in software engineering has been started [3,5,6,12,15,26]. Beyond chatbots or conversational natural language interfacing, natural language interfacing will play an increasingly important and popular role in software development environments [9], due to its benefits of improving developer productivity.

Continuous Learning. Machine learning has been increasingly applied or adapted for various software engineering tasks since at least early 2000 [2,32]; in the past several years, deep learning [24] has been applied on software engineering problems (e.g., [10,11,35]). Such direction's increasing popularity is partly thanks to the availability of rich data (being either explicitly or implicitly labeled in one way or another) in software repositories [32] along with the advances in machine learning especially deep learning [24] in recent years. Beyond applying machine learning only once or occasionally, software engineering tools are in need of gaining the continuous-learning capability: when the tools are applied in software engineering practices, the tools continuously learn to get more and more intelligent and capable.

3 Software Engineering for Intelligence Software

In recent decades, Artificial Intelligence (AI) has emerged as a technical pillar underlying modern-day solutions to increasingly important tasks in daily life and work. The impacted settings range from smartphones carried in one's pocket to transportation vehicles. Artificial Intelligence (AI) solutions are typically in the form of software. Thus, intelligence software is naturally amenable to software engineering issues such as dependability [8] including reliability [22,27] and security [33,34], etc. Assuring dependability of intelligence software is critical but largely unexplored compared to traditional software.

For example, intelligence software that interacts with users by communicating content (e.g., chatbots, image-tagging) does so within social environments constrained by social norms [7]. For such intelligence software to meet the users' expectation, it must comply with accepted social norms. However, determining what are accepted social norms can vary greatly by culture and environment [7]. Recent AI-based solutions, such as Microsoft's intelligent chatbot Tay [16] and Google Photos app's imagine-tagging feature [4], received negative feedback because their behavior lied outside accepted social norms. In addition, formulating failure conditions and monitoring such conditions at runtime may not be acceptable for intelligence software such as robotic software because it would be too late to conduct failure avoidance or recovery actions when such failure conditions are detected [23]. Generally formulating proper requirements for intelligence software remains a challenge for the research community.

In addition, intelligence software is known to often suffer from the "no oracle problem" [19,25,36,37]. For example, in supervised learning, a future application-data entry can be labeled (manually or automatically); however, using such labels as the test oracle is not feasible. The reason is that there exists some inaccuracy (i.e., predicting a wrong label) in the learned classification model. This inaccuracy is inherent and sometimes desirable to avoid the overfitting problem (i.e., the classification model performs perfectly on the training data but undesirably in future application data).

References

1. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proceedings of the Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE), pp. 25–34 (2007)
2. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness, September 2017. arXiv:1709.06182
3. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: learning to write programs. In: Proceedings of the International Conference on Learning Representations (ICLR) (2017)
4. Barr, A.: Google mistakenly tags black people as 'gorillas', showing limits of algorithms. Wall Str. J. (2015). http://blogs.wsj.com/digits/2015/07/01/google-mistakenly-tags-black-people-as-gorillas-showing-limits-of-algorithms/
5. Beschastnikh, I., Lungu, M.F., Zhuang, Y.: Accelerating software engineering research adoption with analysis bots. In: Proceedings of the International Conference on Software Engineering (ICSE), New Ideas and Emerging Results Track, pp. 35–38 (2017)
6. Bieliauskas, S., Schreiber, A.: A conversational user interface for software visualization. In: Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT), pp. 139–143 (2017)

7. Coleman, J.: Foundations of Social Theory. Belknap Press Series. Belknap Press of Harvard University Press, Cambridge (1990)
8. Committee on Technology National Science and Technology Council and Penny Hill Press: Preparing for the Future of Artificial Intelligence. CreateSpace Independent Publishing Platform, USA (2016)
9. Ernst, M.D.: Natural language is a programming language: applying natural language processing to software development. In: Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL), pp. 4:1–4:14 (2017)
10. Gu, X., Zhang, H., Zhang, D., Kim, S.: Deep API learning. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 631–642 (2016)
11. Gu, X., Zhang, H., Zhang, D., Kim, S.: DeepAM: migrate APIs with multi-modal sequence to sequence learning. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), pp. 3675–3681 (2017)
12. Gupta, R., Pal, S., Kanade, A., Shevade, S.: DeepFix: fixing common C language errors by deep learning. In: Proceedings of the National Conference on Artificial Intelligence (AAAI) (2017)
13. Harman, M.: The role of artificial intelligence in software engineering. In: Proceedings International Workshop on Realizing AI Synergies in Software Engineering (RAISE), pp. 1–6 (2012)
14. Jordan, M.: Artificial intelligence-the revolution hasn't happened yet, April 2018. https://medium.com/@mijordan3/artificial-intelligence-the-revolution-hasnt-happened-yet-5e1d5812e1e7
15. Lebeuf, C., Storey, M.D., Zagalsky, A.: How software developers mitigate collaboration friction with chatbots. CoRR abs/1702.07011 (2017). http://arxiv.org/abs/1702.07011
16. Leetaru, K.: How Twitter corrupted Microsoft's Tay: a crash course in the dangers of AI in the real world. Forbes (2016). https://www.forbes.com/sites/kalevleetaru/2016/03/24/how-twitter-corrupted-microsofts-tay-a-crash-course-in-the-dangers-of-ai-in-the-real-world/
17. Li, F.F.: How to make A.I. that's good for people, March 2018. https://www.nytimes.com/2018/03/07/opinion/artificial-intelligence-human.html
18. Michail, A., Xie, T.: Helping users avoid bugs in GUI applications. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 107–116 (2005)
19. Murphy, C., Kaiser, G.E.: Improving the dependability of machine learning applications. Technical report, CUCS-049-, Department of Computer Science, Columbia University (2008)
20. Pandita, R., Xiao, X., Yang, W., Enck, W., Xie, T.: WHYPER: towards automating risk assessment of mobile applications. In: Proceedings of the USENIX Conference on Security (SEC), pp. 527–542 (2013)
21. Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S., Paradkar, A.: Inferring method specifications from natural language API descriptions. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 815–825 (2012)
22. Pei, K., Cao, Y., Yang, J., Jana, S.: DeepXplore: automated whitebox testing of deep learning systems. In: Proceedings of the Symposium on Operating Systems Principles (SOSP), pp. 1–18 (2017)
23. Qin, Y., Xie, T., Xu, C., Astorga, A., Lu, J.: CoMID: context-based multi-invariant detection for monitoring cyber-physical software. CoRR abs/1807.02282 (2018). https://arxiv.org/abs/1807.02282

24. Schmidhuber, J.: Deep learning in neural networks. Neural Netw. **61**(C), 85–117 (2015)
25. Srisakaokul, S., Wu, Z., Astorga, A., Alebiosu, O., Xie, T.: Multiple-implementation testing of supervised learning software. In: Proceedings of the AAAI-2018 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS) (2018)
26. Storey, M.D., Zagalsky, A.: Disrupting developer productivity one bot at a time. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp. 928–931 (2016)
27. Tian, Y., Pei, K., Jana, S., Ray, B.: DeepTest: automated testing of deep-neural-network-driven autonomous cars. In: Proceedings International Conference on Software Engineering (ICSE), pp. 303–314 (2018)
28. Turing, A.M.: Computing machinery and intelligence (1950). One of the most influential papers in the history of the cognitive sciences. http://cogsci.umn.edu/millennium/final.html
29. Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 461–470 (2008)
30. Xiao, X., Paradkar, A., Thummalapenta, S., Xie, T.: Automated extraction of security policies from natural-language software documents. In: Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE), pp. 12:1–12:11 (2012)
31. Xie, T.: Intelligent software engineering: synergy between AI and software engineering. In: Proceedings of the Innovations in Software Engineering Conference (ISEC), p. 1:1 (2018)
32. Xie, T., Thummalapenta, S., Lo, D., Liu, C.: Data mining for software engineering. Computer **42**(8), 55–62 (2009)
33. Yang, W., Kong, D., Xie, T., Gunter, C.A.: Malware detection in adversarial settings: exploiting feature evolutions and confusions in Android apps. In: Proceedings Annual Computer Security Applications Conference (ACSAC), pp. 288–302 (2017)
34. Yang, W., Xie, T.: Telemade: a testing framework for learning-based malware detection systems. In: Proceedings AAAI-2018 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS) (2018)
35. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. In: Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL) (2017)
36. Zheng, W., Ma, H., Lyu, M.R., Xie, T., King, I.: Mining test oracles of web search engines. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 408–411 (2011)
37. Zheng, W., et al.: Oracle-free detection of translation issue for neural machine translation. CoRR abs/1807.02340 (2018). https://arxiv.org/abs/1807.02340
38. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: mining and recommending API usage patterns. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 318–343. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03013-0_15
39. Zhong, H., Zhang, L., Xie, T., Mei, H.: Inferring resource specifications from natural language API documentation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 307–318 (2009)
40. Zhong, Z., et al.: Generating regular expressions from natural language specifications: are we there yet? In: Proceedings of the Workshop on NLP for Software Engineering (NL4SE) (2018)

Software Assurance

Automatic Support of the Generation and Maintenance of Assurance Cases

Chung-Ling Lin[1], Wuwei Shen[1](✉), Tao Yue[2], and Guangyuan Li[3]

[1] Department of Computer Science, Western Michigan University, Kalamazoo, USA
{chung-ling.lin,wuwei.shen}@wmich.edu
[2] Simula Research Laboratory, University of Oslo, Oslo, Norway
tao@simula.no
[3] State Key Laboratory of Computer Science, Institute of Software, Beijing, China
ligy@ios.ac.cn
[4] China School of Computer and Control Engineering, UCAS, Beijing, China

Abstract. One of the challenges in developing safety critical systems is to ensure software assurance which encompasses quality attributes such as reliability and security as well as functionality and performance. An assurance case, which lays out an argumentation-structure with supporting evidence to claim that software assurance in a system is achieved, is increasingly considered as an important means to gain confidence that a system has achieved acceptable safety when checking with emerging standards and national guidelines. However, the complexity of modern safety critical applications hinders the automatic integration of heterogeneous artifacts into an assurance case during a development process such as a V-model, let alone the automatic support of system evolution. In this paper, we present a novel framework to automatically generate assurance cases via safety patterns and further support the maintenance of them during a system's evolution. The application of safety patterns not only enables reusability of previously successful argument structures but also directs the support of assurance maintenance caused by common types of modifications in safety critical domains. The framework is implemented as a prototypical tool built using Model Driven Architecture (MDA). We evaluated the framework with two case studies featuring two criteria and the preliminary experimental results not only show that the framework is useful in evaluation of safety critical systems but also reveal how different types of modification can affect a structure of an assurance case.

Keywords: Assurance case · Safety pattern · Model transformation
Safety critical systems

1 Introduction

Failure of safety critical applications might lead to serious consequences such as significant financial loss or even loss of life. Thus, software quality assurance, also simply called software assurance, has become the focus when certifying a safety critical system. Software assurance includes reliability, security, robustness, safety, and other quality-related attributes, as well as functionality and performance [1]. A structure, like a legal case which lays out an argument structure with supporting evidence, can be helpful to make decisions on software assurance. This kind of structures is called assurance cases or safety cases. An assurance case is thus increasingly considered as an important means to show that a system has acceptable safety when checking its conformance to emerging standards and national guidelines such as the ISO 26262 functional safety standard for automotive systems [2] and the U.S. Food and Drug Administration draft guidance on the production of infusion pump systems [3]. However, most assurance cases are currently manually constructed. For example, the preliminary assurance case for co-operative airport surface surveillance operations [4] is about 200 pages long, and is expected to grow as the interim and operational safety cases are created. The manual management of such increasingly complicated assurance cases are not only time consuming but also error prone. Thus, any automation in the management of assurance cases is appreciated. As such, we in this paper propose an automatic mechanism to create and maintain assurance cases for the purpose of assisting human decisions on software assurance.

One challenge in the automatic construction and maintenance of an assurance case originates from the management of heterogeneous artifacts developed during a software development lifecycle. In many cases, these artifacts are eventually used as evidence in the assurance case. Such heterogeneous artifacts have various formats. The construction of an assurance case including heterogeneous artifacts as evidence as well as their context, assumption and logic inference is further compounded by their underlying development process. Each safety critical domain often has its own development process while they share similarity conceptually.

Efforts have been made to construct assurance cases to support software certification for safety critical systems. Denny et al. proposed a new methodology in the aviation domain to automatically assembly lower level safety cases based on the verification of safety-related properties of the autopilot software in the Swift Unmanned Aircraft System (UAS) [5]. The methodology applied the annotation schema, which models the information such as the definition of a program variable during the verification of safety-related properties, to automatically generate safety case fragments. But, the annotation schema cannot be employed to model system artifacts and high-level safety cases thus are manually assembled. Especially inspired by rapid and successful application of design patterns in software development, researchers have proposed safety patterns in the construction of assurance cases. However, we found most existing approaches based on support of safety patterns lack flexibility when combining two safety patterns in various environment [6–10]. For instance, when combining a node, say leaf node *A*, in one safety pattern with another one, most existing approaches require the leaf node *be* the same as the root node in the second safety pattern. In this case, we call the connection to the first safety pattern via the *A* leaf node *unchanged*. But, our experience in safety critical domains has shown that the *A* leaf node can connect to the root node via the *supportedBy* relation to increase the reusability of the two safety patterns. In this case, we call the connection to the first safety pattern via *A strengthened.*

The automatic management of an assurance case in safety critical domains is a crucial yet feasible feature and can be realized thanks to the development of Model

Driven Architecture (MDA) [12]. As is well known, many safety critical systems adopt a development process such as a V-Model where a sequence of activities are performed and each activity produces a set of artifacts based on the artifacts of its prior activity as input. During a development process, an artifact produced by one activity can become an input to the next activity. Thus, in MDA, using a metamodel, which models a development process, makes it possible to automatically generate a complete assurance case. To this end, we employ the Object Constraint Language (OCL) to expand the syntax of the Goal Structuring Notation (GSN) [13] so that a safety pattern can be defined in a way where more flexibility can be achieved compared with other existing approaches. Furthermore, since only support of the *unchanged* connection to the first safety pattern via a node limits the application of a safety pattern, we add a new feature of having a client to decide whether a connection to the first safety pattern via a node is *unchanged* or *strengthened*.

More importantly, the creation of an assurance case with our framework via safety patterns further alleviates the difficulty in managing the evolution of an assurance case during software maintenance. A root cause for software maintenance commonly comes from two types of modifications, i.e. modifications in a development process, and modifications in artifacts produced for a specific project. In general, if either a development process or a system artifact is modified, a previously laid-out argument structure with supporting evidence might be in jeopardy. Instead of regeneration and review of an entire argument structure, we propose a novel strategy by means of highlighting affected nodes according to modifications. Certifiers can thus concentrate on the affected nodes with their underlying argument structure to recertify a software system. In summary, we make the following contributions in this paper.

- Extension to GSN to allow the introduction of safety case patterns;
- Strengthening an argument structure of a safety pattern when combined with another safety pattern;
- Automatic support of safety case pattern validation and assurance case generation based on the extension to GSN to increase the reusability of successful safety case patterns so that the capability of building and reviewing a safety case can be improved; and
- Integration of safety case generation and maintenance into a development process.

The remainder of this paper is organized as follows: In Sect. 2, we briefly introduce the GSN and present a problem statement for this research. In Sect. 3, we introduce a running example to illustrate our framework. Section 4 discusses the framework structure and algorithm with its application in one case study. Section 5 presents the evaluation while Sect. 6 discusses the related work. We draw a conclusion and present the future work in Sect. 7.

2 GSN and the Problem Statement

An assurance case is important for safety critical systems in that it provides an argument structure about why a safety critical system achieves its mission. Currently, there are two notations to denote an assurance case, i.e. GSN and Claims-Arguments-

Evidence (CAE) Notation [14]. In this paper, we consider GSN as the main representation of an assurance case. Central to an assurance case is an argument that is given by a hierarchy of claims supporting a logic reasoning chain to support an overall claim. To provide software assurance of a safety-critical system, a convincing assurance case usually consists of claims, strategies, and evidence as well as assumptions and justifications, which is illustrated via a GSN example shown in Fig. 4 with a brief introduction as follows.

- Claim. A claim, also called a goal, is a statement that claims about some important properties of a system such as safety and security. In GSN, a claim of an argument is documented as a goal which is denoted as rectangle. In Fig. 4, for instance, the top rectangle node "*BSCU SW is safe*" is a claim or a goal.
- Strategy. When a claim is supported by a set of sub-claims, it is useful to provide the reasoning step as part of the whole argument. This reasoning step is called a strategy. In GSN, a strategy links between two or more goals and is rendered as a parallelogram. In Fig. 4, for instance, the parallelogram node "*Argument over all BSCU software contributions identified from BSCU SW"* is a strategy.
- Evidence. Evidence is used to support the truth of a claim. In GSN, evidence is documented as a solution and represented as a circle. In Fig. 4, for instance, the circle node "WBS Hazard Analysis Report" is an evidence item.

A convincing and valid argument regarding why a system meets its assurance properties is given by a logic reason chain which is the heart of an assurance case. A line with solid arrowhead between two nodes in GSN denotes the "*supportedBy*" relationship. For instance, the top claim of the GSN in Fig. 4 is supported by the strategy, which is the argument over all "BSCU hazards" being identified from "BSCU SW". The strategy is further supported by four subclaims, three of which show the three hazards while the fourth shows all these three hazards are completely and correctly identified. Each subclaim can be further supported until some evidence node is provided. For instance, the fourth subclaim is further decomposed into another subclaim about the "Functional Hazard Analysis" method. Finally, the subclaim is supported by an evidence node showing the WBS hazard analysis report. Due to space, we skip the further argumentation structure supporting the first three subclaims related to the identified hazards using a diamond, meaning *undeveloped*. Once a GSN model is established, a certifier can review the model in a bottom-up manner to gain the confidence of a root claim.

The goal of this paper is to aid both engineers and certifiers to develop and certify a safety critical system in an effective and efficient manner. We thus address the following problem statement: *Given a specific development process employed by engineers, while software assurance is finally made by a human judgement of fitness for use, how can we provide as much automation as possible to generate an assurance case which not only lays out an argument structure with supporting evidence to support a claim about software assurance but also evolves during software maintenance*. To attack the challenging problem statement, we will present a novel framework based on MDA [12]. The framework provides the automation of the creation and maintenance of an assurance case in support of development and certification of safety critical systems.

3 An Illustrative Example

To illustrate how our framework supports development and certification of safety critical systems, we use a Wheel Braking System (WBS) for an aircraft called the S18 based on the system specification in the Aviation Recommended Practice (ARP) 4761 Appendix L. To demonstrate the capability of our approach in support of automatic generation of an assurance case, we use the system artifacts which were provided by [6], where the authors manually generated an assurance case. An important reason to choose this case study is that the case study was well developed and discussed by a third-party developer team and so any bias when applying our approach can be removed.

According to the work [6], there are two independent Brake System Control Units (BSCUs). To demonstrate that a software system is sufficiently safe to operate on the BSCU, all hazards should be first identified and then hazardous software contributions related to the identified hazards should be revealed. For each hazardous software contribution, a set of safety requirements are presented in [6]. All this information can be modelled as a class diagram in the Unified Modeling Language (UML), called a domain model. Specifically, a domain model provides the meta-information which is required for a specific project. In many cases, a domain model can be converted to the meta information representing the content in a table. For instance, the classes *Hazard*, *SWContribution*, *SoftwareSafetyReq* are converted to the three columns in Table 1. For a specific system, the values/instances of the classes in a domain model are listed in the table. The authors in [6] listed three hazards, three hazardous software contributions, and three software safety requirements which are summarized in Table 1.

Table 1. Example of system artifacts

Software	Hazard	SWContribution	SoftwareSafetyReq
BSCU SW	Hazard 1 Loss of Deceleration Capability	Contribution 1 Software fails to command braking when required	SSR1 On receipt of brake pedal position. Command shall calculate braking force and output braking command
	Hazard 2 Inadvertent Deceleration	Contribution 2 Software commands braking when not required	SSR2 Command shall not output a braking command unless brake pedal position is received
	Hazard I Partial Loss of Deceleration Capability and Asymmetric Deceleration	Contribution 3 Software commands incorrect braking force	SSR3 Braking commands generated by the Command component meet defined criteria

Figure 1 shows a safety pattern catalog whose pattern in the extended GSN is supported by the framework. Unlike many existing safety pattern-based approaches, we propose that a safety pattern catalog is necessary for the application of the safety pattern in that the catalog provides not only a safety pattern in the extended GSN but also includes the description and applied metamodel to which the pattern can be applied. Therefore, a safety pattern can clearly demonstrate how it is applied in a concrete environment via an applied metamodel combined with the description. As shown in Fig. 1, we extend the syntax of GSN to accommodate the introduction of new variables. Basically, there are two types of variables. One is given by the $\{s\}$ variable where s is mapped to a class in an applied metamodel. The other type is given by $\{\$a\}$ where variable $\$a$ denotes a string during the instantiation of a pattern. So, when the safety pattern shown in Fig. 1 is applied to the metamodel with the configuration table as shown in Fig. 3(2), an assurance case can be generated via the safety pattern where the artifacts are produced.

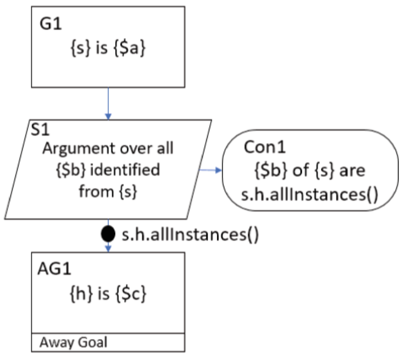
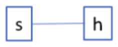
Pattern Description	Safety Pattern in GSN	Applied Metamodel
The general argument pattern: To claim {s} satisfies a property {$a} (e.g. safe/addressed), we argue that all identified properties {$b} (class mapped by {h}) of {s}, which are derived from {s.h.allInstances()}, satisfy property {$c}	 G1 {s} is {$a} S1 Argument over all {$b} identified from {s} Con1 {$b} of {s} are s.h.allInstances() s.h.allInstances() AG1 {h} is {$c} Away Goal	

Fig. 1. Safety pattern catalog – Pattern 1

Based on some artifacts produced as an exemplary of a particular system as shown in Fig. 3(1), an assurance case is generated as shown in Fig. 3(4) as an application of the pattern shown in Fig. 1 via our framework. The advantage of the framework is that as more instances of the classes *Hazard*, *SWContribution*, and *SoftwareSafetyReq* are created, the assurance case is augmented according to the safety pattern and its configuration table. In our framework, the language for specifying a safety pattern has the following features. First, a link relationship between some artifacts makes the structure of an assurance case appropriately to be set up. In Fig. 3(4), for instance, three sub-claims, which are related to three hazards instances which are associated to the BSCU software, are generated to support the strategy node "*Argument over all BSCU hazards identified from BSCU SW*". Second, the application of string variables which are replaced with a string value during the instantiation of a pattern provides more flexibility in support of various types of claims. In Fig. 1, $a denotes a property variable and a real property name depends on the real scenario and, for example, $a is replaced with "safe" in Fig. 3(4). String variables increase the flexibility of a safety pattern in

Pattern Description	Safety Pattern in GSN	Applied Metamodel

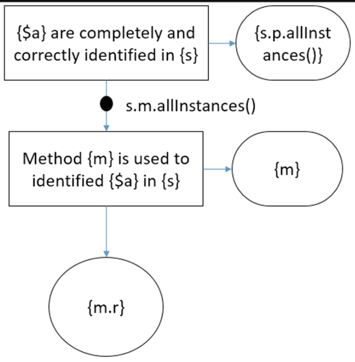
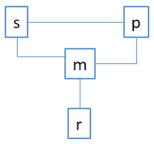
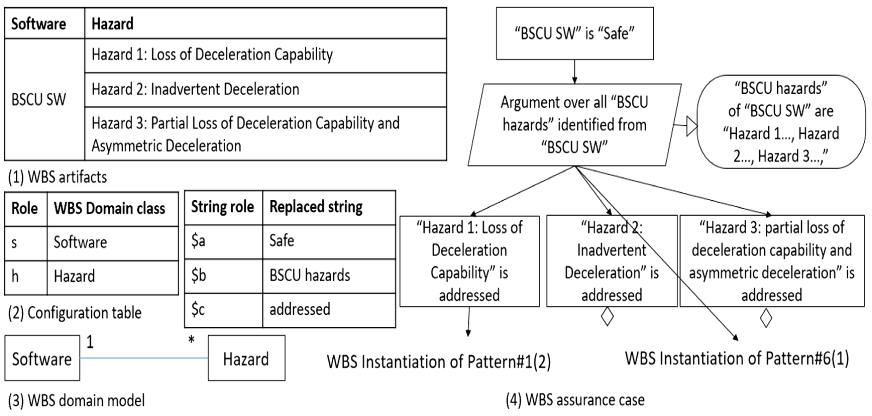
Identification pattern: The claim that property {$a} (class mapped by {p}) of {s}, which are derived from {s.p.allInstances()}, are completely and correctly identified in {s}. This claim is supported by methods {s.m.allInstances()} used in the identification process, and the evidence of the identification process is given by {m.r}	 {$a} are completely and correctly identified in {s}
	{s.p.allInstances()}
	s.m.allInstances()
	Method {m} is used to identified {$a} in {s}
	{m}
	{m.r}
		

Fig. 2. Safety pattern catalog – Pattern 2

that $a can be replaced with some other claims such as secure provided that a new assurance case uses the same argument structure of the safety pattern.

The leaf nodes of the assurance case shown in Fig. 3(4) address the three WBS hazards respectively. These three claims should be further supported by three goal nodes, each of which claims that a corresponding software contribution related to a hazard is addressed in a separate assurance case. In this case, each goal node matches the root node of pattern 1 and the subtree under each goal node is generated via the application of pattern 1. In this case, the connections to the first safety pattern via the three goal nodes are all unchanged. Due to space, we only show how node *G2* is supported in Fig. 4. However, in another scenario, the connection to the first safety pattern via a node can be strengthened. For instance, the strategy in Fig. 3(4) can be further strengthened by the claim that all identified hazards are correct and complete, and no other hazard exists. Thus, pattern 2 in Fig. 2 is connected to pattern 1 via the strategy node in Fig. 3(4). Specifically, in the WBS assurance case shown in Fig. 4, the claim node (i.e. node G5) *"BSCU hazards are completely and correctly identified in BSCU SW"* as an application of pattern 2, is connected to the strategy node (i.e. node S1) as an application of pattern 1 so the argument structure of pattern 1 is strengthened. To make safety patterns more reusable, we add a new feature of having a client to decide whether a connection to a first safety pattern via a node is unchanged or strengthened.

On the other hand, modification of system artifacts cannot be avoided and the modification thus can undermine the validity of a previously laid-out argument structure. For instance, if the Hazard 1 in the WBS example is changed, then all the corresponding nodes in the assurance case in Fig. 4 are affected and thus marked as ①. Furthermore, all the nodes along the path from an affected node upwards to the root are also undermined and thus we mark them as ②. Next, the traceability information further helps to find affected nodes. Specifically, since Hazardous Software Contribution 1 is derived from Hazard 1 via the traceability information, all the nodes in the



Software	Hazard
BSCU SW	Hazard 1: Loss of Deceleration Capability
	Hazard 2: Inadvertent Deceleration
	Hazard 3: Partial Loss of Deceleration Capability and Asymmetric Deceleration

(1) WBS artifacts

Role	WBS Domain class
s	Software
h	Hazard

(2) Configuration table

String role	Replaced string
$a	Safe
$b	BSCU hazards
$c	addressed

Software
1
*
Hazard
(3) WBS domain model
"BSCU SW" is "Safe"
Argument over all "BSCU hazards" identified from "BSCU SW"
"BSCU hazards" of "BSCU SW" are "Hazard 1..., Hazard 2..., Hazard 3...,"
"Hazard 1: Loss of Deceleration Capability" is addressed
"Hazard 2: Inadvertent Deceleration" is addressed
"Hazard 3: partial loss of deceleration capability and asymmetric deceleration" is addressed
WBS Instantiation of Pattern#1(2)
WBS Instantiation of Pattern#6(1)
(4) WBS assurance case

Fig. 3. An assurance case

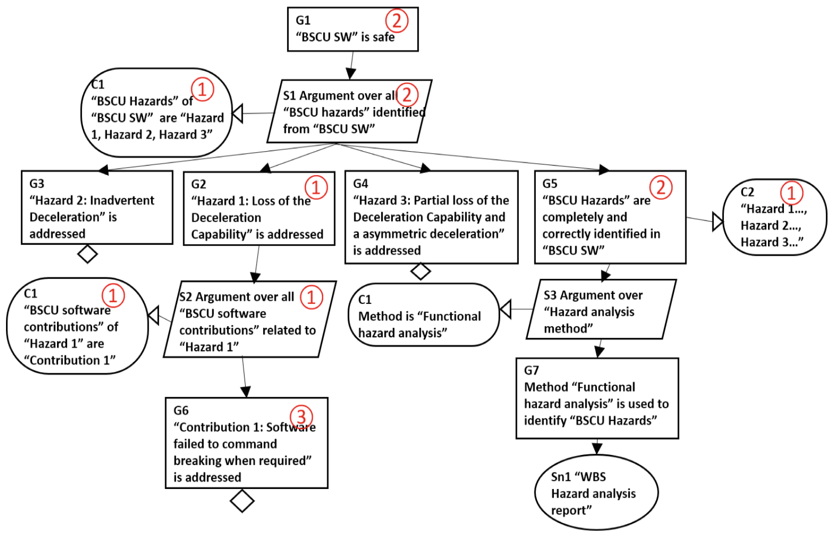

G1 "BSCU SW" is safe 2
S1 Argument over all "BSCU hazards" identified from "BSCU SW" 2
C1 "BSCU Hazards" of "BSCU SW" are "Hazard 1, Hazard 2, Hazard 3" 1
G3 "Hazard 2: Inadvertent Deceleration" is addressed
G2 "Hazard 1: Loss of the Deceleration Capability" is addressed 1
G4 "Hazard 3: Partial loss of the Deceleration Capability and a asymmetric deceleration" is addressed
G5 "BSCU Hazards" are completely and correctly identified in "BSCU SW" 2
C2 "Hazard 1..., Hazard 2..., Hazard 3..." 1
C1 "BSCU software contributions" of "Hazard 1" are "Contribution 1" 1
S2 Argument over all "BSCU software contributions" related to "Hazard 1" 1
C1 Method is "Functional hazard analysis"
S3 Argument over "Hazard analysis method"
G7 Method "Functional hazard analysis" is used to identify "BSCU Hazards"
G6 "Contribution 1: Software failed to command breaking when required" is addressed 3
Sn1 "WBS Hazard analysis report"

Fig. 4. WBS assurance case

assurance case which are linked to Hazardous Software Contribution 1 are highlighted as ③ in Fig. 4. Numbers within a circle denote the order of how nodes in an assurance case are affected and reported. As a result, certifiers should review all the affected nodes in the assurance case.

4 The Framework Overview and Its Main Algorithms

4.1 Overview of the Framework

Our framework includes four main activities, shown in Fig. 5. First, the Instantiable Validation activity (1) determines whether a generic safety pattern can be applied to a domain model via configuration tables. This activity ensures that all variables in a safety pattern can be appropriately instantiated. Second, the Variable Replacement Activity (2) aims to generate a domain specific pattern by replacing all variables in a safety pattern with domain classes via a configuration table. Third, the Pattern Combination activity (3) employs pattern connection tables to generate a composite domain specific pattern via the combination of the generic safety patterns. As a result of activities (1), (2) and (3), a complete domain specific pattern is generated using an ATL (Atlas Transformation Language) program [15]. For a specific project, the ATL program takes all artifacts, which are a valid instance of a domain model, as input and generates an assurance case as output. Once an assurance is generated, the Maintenance Activity (4) takes over to monitor any modification in a domain model or any artifacts related to the project. Once, a modification is found, the maintenance activity carries out the impact calculation to figure out how the previous assurance case is affected by highlighting the affected nodes in the assurance case.

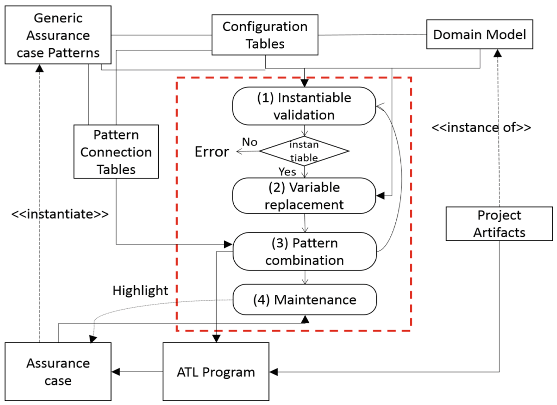


Fig. 5. Flow chart of our framework

4.2 Generation of Assurance Case and Its Algorithms

To finally produce an assurance case for a project in a specific domain model via the application of various safety patterns, the generation process consists of two steps. First, a complete safety pattern for a specific domain is produced and this is called a complete domain specific safety pattern, or just a complete safety pattern for simplicity. In a complete safety pattern, each variables is replaced with a domain class; for instance, variable s is replaced with domain class *Hazard*. As output of the first step, the framework produces an ATL program ready to output a specific assurance case based on the complete safety pattern. Second, to produce a final assurance case for a specific project, we run the generated ATL program using the Eclipse ATL framework, shown in the bottom of Fig. 5 and the ATL program uses the pattern connection information to connect the second safety pattern to the first safety pattern via a node.

We outline the algorithm for the first step to generate a complete safety pattern in Fig. 6. The algorithm takes a configuration table as input given by the *c* parameter and all pattern connection information, given by the *PC* parameter. This algorithm generates an ATL program which is ready to produce an assurance based on a complete safety pattern such as the one in Fig. 8(2). Note that a configuration table includes a safety pattern to be instantiated. Our framework supports two types of pattern connections based on the type of a node involved in the connection to build a complete safety pattern. The first type, where the connection to a first safety pattern via a node is unchanged, combines two patterns via the duplicate nodes between the two patterns. This type of connection requires the identical nodes generated in the first pattern and a starting node of the second pattern.

```
Create a complete domain specific pattern
Input: The first configuration table used in the project, list of all Pattern Connection PC
Output: An project specific combined assurance pattern newp

1   createCombinedPattern(c, PC)
2   {
3       Pattern newp = blank
4       identify the pattern c.p defined in c, and the root node n of c.p
5       addNode(newp, c.p, n, c, PC)
6   }
7   // newp a domain specific pattern to be generated
8   // p: current safety pattern; n: the root node of p;
9   // c: current configuration table; PC: a list of all pattern connections;
10  addNode(newp, p, n, c, PC)
11  {
12      for each variable v in n
13          if v is instantiable
14              replace v to dc via c
15      add node n to newp
16      for each link l in p   // for links in the same pattern p
17          if n = l.sn
18              add l to newp
19              addNode(newp, p, l.tn, c, PC)
20      List<PatternConnection> pcs = checkPatternConnection(p, PC, c, n)
21      if (pcs != null)   // node n has link(s) to other pattern(s)
22      for each PatternConnection pc in pcs
23          identify the root node rn of pc.tp.p
24          if type of n is away goal // first type of combination
25              if n is identical to rn
26                  find the parent node pn of n and link pl between pn and n
27                      link nl = new link(pn, rn, pc.m)  // create a new link for pn to rn
28                      remove n and pl from newp // remove the duplicate node
29              else  // second type of combination
30                  link nl = new link(n, rn, pc.m)  // create a new link for n to rn
31              add nl to newp
32              addNode(newp, pc.tp,  rn, pc.tc, PC)
33      }
34  List<PatternConnection> checkPatternConnection(p, PC, c, n)
35  {
36  List<PatternConnection> pcs = blank // the matched pattern connections
37  for each Pattern Connection pc in PC
38  if ( p = pc.sp.p AND n = pc.n AND c = pc.sc)
39          remove pc from PC // update the pattern connections
40          add pc to pcs
41  return pcs
42  }
```

Fig. 6. Pattern combination algorithm

Figure 8 illustrates the generation of a complete safety pattern shown in Fig. 8(2) via the connection of a unchanged node *N1.3* between two patterns shown Fig. 8(1). Specifically, the complete safety pattern shown in Fig. 8(2) is generated by connecting pattern 1 to itself via duplicating nodes *N1.3* and *N1.1* in pattern 1 (shown in Fig. 8(1)). So, the algorithm in Fig. 6 starts from generating an empty pattern denoted by variable *newp* at line 3. Next, it calls method *addNode()* at line 5 in Fig. 6 on the root node *N1.1* of *p1* identified at line 4. Node *N1.1* is added to cp as *N1.1* by line 15. Next, all links are added to cp, i.e. the link from *N1.1* to *N1.2* in *p1*, by line 18 and method *addNode*() is called on *N1.2* by line 19 to add *N1.2* with its all links to cp. Line 20 checks if *N1.1* is linked to another pattern via method *checkPatternConnection()* from line 34 to line 42 in Fig. 6. Since there is no link between *N1.1* to any other pattern, method *checkPatternConnection()* returns an empty list, and thus the *addNode()* method called

Configuration Table: c1 (Pattern1)	
Role	**WBS Domain class**
s	Hazard
h	SWContribution

Configuration Table: c2 (Pattern2)	
Role	**WBS Domain class**
s	Software
p	SWContribution

Configuration Table: c3 (Pattern1)	
Role	**WBS Domain class**
s	SWContribution
h	SafetyRequirement

(1) Configuration tables

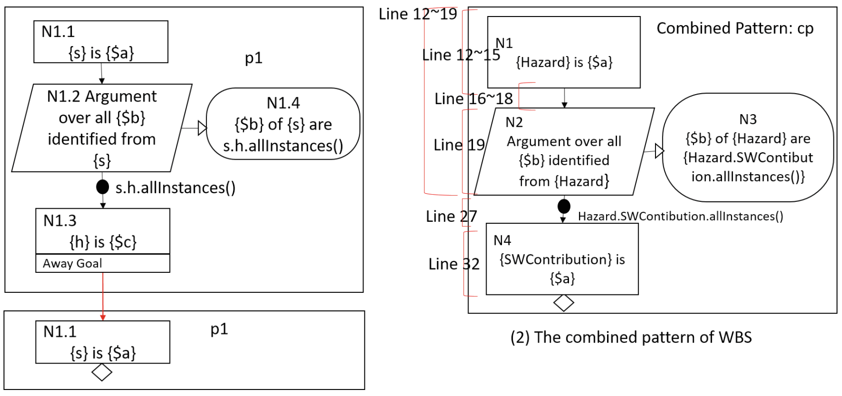
Source Pattern	Source Node	Source Pattern Configuration Table	Target Pattern	Target Pattern Configuration Table	Multiplicity
p1	N1.2	c1	p2	c2	1
p1	N1.3	c1	p1	c3	

(2) Pattern connection

1. LI = {"is", "Argument over all", "identified from", "are", "are completely and correctly identified in"}
2. V = { s, $a, $b, h, s.h.allInstances(), p, s.p.allInstances()}
3. N1.1 = < {"is"}, {s, $a} >, N1.2 = <{"Argument over all", "identified from"}, {$b, s}>, N1.3 = < {{h, $c}}>, N1.4 = <{"of","are"},{$b,s,s.h.allInstances()}>, N2.1 =<{"are completely and correctly identified"},{s,$a}>, N2.2 = <{},{s.p.allInstances()}>. VN = {N1.1, N1.2, N1.3, N1.4, N2.1, N2.2}
4. N = {}
5. l1 = <N1.1, N1.2, 1>, l2 = <N1.2, N1.3, s.h.allInstances()>, l3 = <N1.2, N1.4, 1>, l4 = <N2.1, N2.2, 1> , L = { l1, l2, l3, l4 }
6. vn1 = {N1.1, N1.2, N1.3, N1.4}, vn2 = {N2.1, N2.2}, p1 = <vn1, {l1, l2, l3}>, p2 = <vn2, {l4}> P = { p1, p2 }
7. DC = {Hazard, SWContribution, Software, SafetyRequirement}
8. mp1 = <s, Hazard>, mp2 = <h, SWContribution>, mp3 = <s, Software>, mp4 = <p, SWContribution>, mp5 = <s, SWContribution>, mp6 = <h, SafetyRequirement>, MP = {mp1,mp2,mp3,mp4,mp5,mp6}
9. c1 = <p1, {mp1, mp2}>, c2 = <p2, {mp3, mp4}>, c3 = <p1, {mp5, mp6}>, C = {c1, c2, c3}
10. pc1 = <c1, c2, N1.2, 1>, pc2 = <c1, c3, N1.3, >, PC = {pc1, pc2}

(3) Data types

Fig. 7. Pattern combination inputs


N1.1
{s} is {$a}
p1
N1.2 Argument over all {$b} identified from {s}
N1.4
{$b} of {s} are s.h.allInstances()
s.h.allInstances()
N1.3
{h} is {$c}
Away Goal
N1.1
{s} is {$a}
p1
Line 12~19
Line 12~15
Line 16~18
Line 19
Line 27
Line 32
Combined Pattern: cp
N1
{Hazard} is {$a}
N2
Argument over all {$b} identified from {Hazard}
N3
{$b} of {Hazard} are {Hazard.SWContibution.allInstances()}
Hazard.SWContibution.allInstances()
N4
{SWContribution} is {$a}

(1) The original patterns

(2) The combined pattern of WBS

Fig. 8. Pattern combination example (via first type pattern connections)

on node *N1.1* is returned. So is the method call *addNode()* on node *N1.2*. When method *addNode* is called on node *N1.3* in the first application (Fig. 8(1)), node *N1.3* is temporary added to the complete safety pattern *cp* by line 15 since a safety pattern is connected to *N1.3*. Line 20 retrieves the information of a second pattern returned by method *checkPatternConnection()* from line 34 to line 42. After identifying the nodes *N1.3* and *N1.1* are the same using line 23 to line 25, the parent node of *N1.3*, i.e., *N1.2*, is identified by line 26, and a new *supportedBy* link between *N2(N1.2)* and *N4(N1.1)* is generated by line 27. After the new link is generated, the duplicate goal node *N1.3* is removed from the complete safety pattern *cp* by line 28. A new link between *N2* and *N4* is added to cp by line 31.

During the combination of two safety patterns, each variable is replaced with a domain class simultaneously. Specifically, in Fig. 8(2), variable s is replaced with domain class *Hazard* in the first application of pattern 1 while variable s is replaced with domain class *SWContribution* in the second application of pattern 1 vis the configuration tables as shown in Fig. 7(1). When the *addNode()* method is called on a node, line 14 replaces a variable with a domain class via the configuration table *c*. When a complete safety pattern is generated via a connection of duplicate nodes, our framework ensures that the two variable in the two duplicate nodes are mapped to the same domain class. For example, *h* in *N1.3* in the first application of pattern 1 is mapped to domain class *SWContribution*, and *$c* is replaced by string "addressed". Variable *s* in *N1.1* in the second application of pattern 1 is also mapped to domain class *SWContribution,* and *$a* is also replaced by "addressed". In this case, both nodes are identical and the combination process continues.

The second type of connection is to strengthen a node in a first safety pattern when connected to a second safety pattern. In this case, a node of the first pattern is different from a starting node/root claim of the second pattern during combination. For example, Fig. 9 shows how two safety patterns are combined to generate a complete domain specific safety pattern via the second type of connection. In this example, the complete safety pattern is generated via the connection between node *N1.2* in the first pattern and starting node *N2.1* of the second pattern (shown in Fig. 9(1)). The copy of the first safety pattern to a complete safety pattern is similar to the previous type so we skip the process. For the second type of connection, the difference is that a new *supportedBy* link between node *N2(N1.2)* and *N5(N2.1)* is added in the complete pattern in Fig. 9(2) and no node is removed. To do so, when the *addNode()* method is called on node *N1.2*, a new *supportedBy* link from *N1.2* in a first pattern to the root node *N2.1* in a second pattern is created by line 30 in Fig. 6. After that, the *addNode()* method adds the link between the corresponding nodes *N2* and *N5* to the complete safety pattern at line 31. Last, method *addNode()* is called on the rest of nodes to complete the combination at line 32. As an example, the top of the assurance case in Fig. 4 is generated by the complete safety pattern in Fig. 9(2) via running the corresponding ATL program.

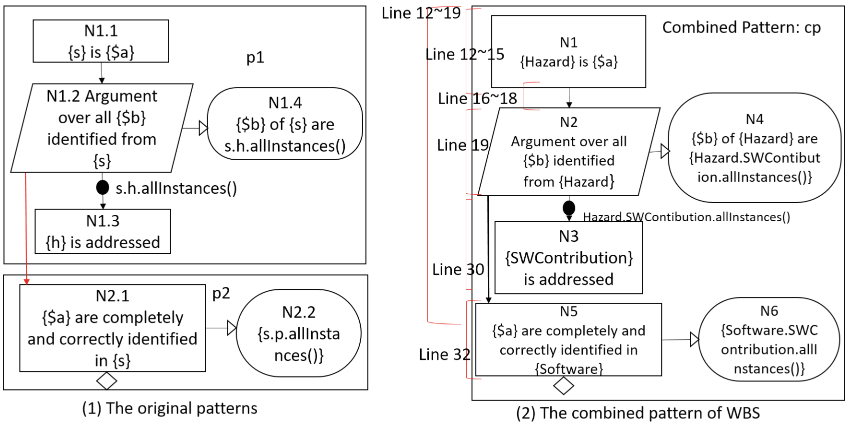
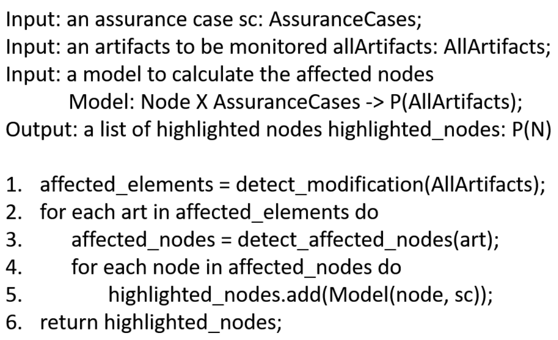

N1.1
{s} is {$a}
p1
N1.2 Argument over all {$b} identified from {s}
N1.4
{$b} of {s} are s.h.allInstances()
s.h.allInstances()
N1.3
{h} is addressed
N2.1
{$a} are completely and correctly identified in {s}
p2
N2.2
{s.p.allInstances()}
(1) The original patterns
Line 12~19
Line 12~15
Line 16~18
Line 19
Line 30
Line 32
Combined Pattern: cp
N1
{Hazard} is {$a}
N2
Argument over all {$b} identified from {Hazard}
N4
{$b} of {Hazard} are {Hazard.SWContibution.allInstances()}
Hazard.SWContibution.allInstances()
N3
{SWContribution} is addressed
N5
{$a} are completely and correctly identified in {Software}
N6
{Software.SWContribution.allInstances()}
(2) The combined pattern of WBS

Fig. 9. Pattern combination example (via second type pattern connections)

4.3 Maintenance of an Assurance Case and Its Algorithm

Next, we introduce another important module in the framework which supports the evolution of a safety critical system. Modification has become an important activity during a software development process. After modifications occur, an assurance case produced by the previous assurance case generation module should be re-considered. Obviously, re-evaluation of an entire assurance case is costly and impractical. A better way to support maintenance of an assurance case during the evolution of a safety critical system is to highlight nodes in the argument structure potentially undermined by a modification.

The algorithm for maintenance of an assurance case is outlined in Fig. 10. The maintenance algorithm takes as input an assurance case previously produced, all artifacts to be monitored, and a computation model to detect affected nodes in an assurance case. The algorithm uses another method *detect_modification* on line 1 which returns a list of modified artifacts. Next, the algorithm is enumerating all modified artifacts starting from line 2. For each modified artifact, the algorithm calls method *detect_affected_nodes* to find a list of nodes in an assurance which are affected by the modified artifacts. The algorithm then employs the computation model provided as input to calculate a list of affected nodes. The returned list of the affected nodes are added to variable *highlighted_nodes* which is finally returned as output of the algorithm.



```
Input: an assurance case sc: AssuranceCases;
Input: an artifacts to be monitored allArtifacts: AllArtifacts;
Input: a model to calculate the affected nodes
        Model: Node X AssuranceCases -> P(AllArtifacts);
Output: a list of highlighted nodes highlighted_nodes: P(N)

1.  affected_elements = detect_modification(AllArtifacts);
2.  for each art in affected_elements do
3.      affected_nodes = detect_affected_nodes(art);
4.      for each node in affected_nodes do
5.          highlighted_nodes.add(Model(node, sc));
6.  return highlighted_nodes;
```

Fig. 10. Maintenance algorithm

While the judgement of software assurance is a decision finally made by humans, some effort has been made to automatically identify the areas where a decision must be rendered. Specifically, various models have been proposed to calculate the confidence of the root node in an assurance case. While various calculations have been carried out, most existing approaches use a model to calculate the confidence of nodes affected by the evidence which is linked to system artifacts. They then set up a threshold to show and determine which nodes may be below the threshold. These nodes confidence is then in doubt. Therefore, the confidence of all nodes along the path to the root of an assurance is reduced and the software assurance about the system is in jeopardy. If the confidence of a node is not below the threshold, then the node's confidence is not reduced. This implies it should not affect the confidence of the other nodes in an assurance case. To this end, we take a most conservative model in this paper where once the framework detects a node whose linked artifact is modified, then the framework highlights all nodes along the path from this node to the root node of the assurance case. The rational behinds this idea comes from the following: To support the evolution of an assurance case, we propose that any miss of highlighting nodes whose confidence is in jeopardy is a costly mistake when evaluating software assurance of a system. Therefore, the system should highlight all potential nodes whose confidence could be undermined by modifications.

5 Evaluation

We implemented our approach in a tool which we applied to two case studies: the Wheel Braking System (WBS) for an aircraft called the S18 [6] and the two tanks control system [16]. For an objective evaluation of our approach, we concentrate on two criteria of the framework. The first criterion is to study how the creation of an assurance case aids the finding of an incomplete argument structure in support of software assurance. As such, we build a new feature in the tool which attempts to find the leaf goal nodes which are not supported by an evidence node. Note a complete assurance case is a tree structure argument where each leaf goal node is supported by an evidence node; otherwise an assurance case has an incomplete argument structure to support a goal node.

Table 2. Assurance case evaluation results

Experiment	Assurance case	Nodes	Unsupported nodes
1	WBS	112	6
2	WBS	109	7
3	Two tanks system	1160	78

In experiment 1 shown in Table 2, our tool generated 112 nodes in the WBS assurance case and identified that 6 goal nodes are not supported by any other node in the WBS assurance case. After the manual check of the WBS assurance case, we found that the 6 unsupported goal nodes are related to safety requirement *SSR2* and *SSR3* shown in Table 1, and the *WBS ABS*, *Input*, *Output*, and *Monitor* modules. And these artifacts are not further considered/traced to any other WBS artifact in [6]. To further test the correctness of our tool for the WBS case study, in experiment 2 shown in Table 2, we temporary remove the *Braking unit testing result* from WBS system artifacts. Our tool generates 109 nodes in the WBS assurance case and identifies one new node related to *Braking Module* is not supported by any node in the WBS assurance case with totally 7 goal nodes not supported by any other node. In experiment 3, our tool generates 1160 nodes in the two tanks control system assurance case and identifies that 78 goal nodes are not supported by any node in the two tanks control system assurance case. The manual check of the assurance case for the two tanks control system indicates that the missing goal nodes in the assurance case are caused by the incomplete traceability information of the system artifacts. These experiment results demonstrate that our tool generates the assurance case based on existing system artifacts, and the evaluation feature of our tool can identify the structural errors of a generated assurance case.

The second criterion is to investigate how a modification during system evolution has an impact on an assurance case from two aspects. The first aspect is, as mentioned before, the two types of modifications, i.e., (1) modification in a development process, or (2) modification in a system artifact of a specific project. The experiment results are summarized in Table 3. From the data in Table 3, we conclude that a development process modification can lead to more affected nodes than its corresponding system artifact modification.

Table 3. Assurance case maintenance results

Experiment	Assurance case	Type of modification	Modified object	Affected nodes
4	WBS	Domain class	Report	60
5	WBS	System artifact	Semantic analysis report	31
6	WBS	System artifact	SSR1	38
7	WBS	System artifact	Braking module source code	9
8	Two tanks system	Domain class	Verification report	133
9	Two tanks system	System artifact	System level requirement analysis report	32
10	Two tanks system	System artifact	Tank 1 architecture model verification report	22

To confirm the conclusion, we manually investigate a structural relationship between the development process used in [6], system artifacts produced by [6], and their assurance case. Specifically, we chose a domain classes *Report* as a subject. There is no node directly linked to *Report* in the generated assurance case. Next, we consider all instances of domain class *Report* which has three instances: *Semantic analysis report*, *Integration test report*, and *Fault Tree Analysis report.* After further manual review of the assurance case, we find that 31 nodes are linked to the system artifact *Semantic analysis report,* 22 nodes to the system artifact *Integration test report*, and 16 nodes to the system artifact *Fault Tree Analysis report*. Note when a modification on an artifact is caught, the affected nodes include all nodes in an assurance case which are directly linked to the artifact as well as the nodes which are linked to all instances of the artifacts. For instance, a modification on "*Semantic analysis report"* occurs in experiment 5, it only affects 31 nodes since they are linked to the report and there is no other instance of the report in the WBS case study. However, when a modification on *Result* occurs in experiment 4, we consider all directly nodes linked to *Result* in the assurance case, i.e., 0 in this case, and then all nodes linked to the three instances of *Report.* In this case, for the three instances, the affected node numbers are 31, 22, and 16 respectively. Since some nodes are linked to multiple instances of *Report*, there are totally 60 nodes which are affected by the modification on *Result*. And a modification on *Result* thus has more affected nodes than a modification on any of an instance of *Result*. Likewise, we did experiments 8,9 and 10 for the two tanks case study and draw the same conclusion.

Next, we consider the second aspect: how a modification during different phases of a SDLC for a project affects an assurance. For the WBS case study, experiment 6 is considered as a modification in an early phase of SDLC compared with experiment 7, a late phase modification. For the two tanks control system case study, experiment 9 is considered as an early phase modification compared with experiment 10. Both experiment sets show that a modification made in an early phase of SDLC affects more nodes in an assurance case compared to a modification made in a late phase of SDLC. A manual check further confirms this conclusion. For instance, a modification of SSR1 affects two system artifacts, Hazard 1 and Contribution 1 via traceability. Artifacts Hazard 1 and Contribution 1 together have 17 corresponding nodes in an assurance case, and SSR1 has 32 corresponding nodes. But among the 49 nodes, 7 nodes reference multiple system artifacts and 4 nodes are affected by the same child nodes in the assurance case, and 38 nodes are thus affected by the modification of SSR1. But artifact "*Braking Module Source code*" has one system artifacts, *Braking Module*, via traceability. And system artifact *Braking Module* has 8 corresponding nodes in the assurance case, and *Braking Module Source code* has 9 corresponding nodes including the 8 corresponding nodes of *Braking Module*. So, the modification of a system artifact generated during the early phase of a SDLC affect more nodes in an assurance case compared with the modification of a system artifact generated during the later phase of a SDLC.

6 Related Work

The insurance of software property assurance in safety critical systems has been a hot topic due to the high demand of safety critical systems in our daily life. An even more compelling argument can be made in safety-critical applications. Many challenges remain in the assurance case community [17]. Various techniques have been proposed to address the problems and difficulties in the construction of an assurance case. Some researchers have presented the application of patterns in building an assurance case.

Hawkins et al. proposed a Model-Based approach to weave an assurance case from design [11]. They used a reference model to model all artifacts and their relationship for a specific project and a GSN metamodel to denote an assurance case. When a pattern is instantiated, a weaving table is used to generate a specific assurance case. But the simple mapping relationship cannot generate a correct underlying reasoning chain in an assurance case.

Denney et al. provided a tool to support assurance-based software development [8]. In their tool, they also proposed a new pattern language to generate an assurance case. Their approach required a data table to connect system artifacts and variables in a safety pattern during its instantiation. Obviously, the mapping relationship between all system artifacts and variables would be manually provided. So, the mapping relationship is a project-dependent input. However, our approach takes the mapping relationship from a domain model to model a development process. Thus, our framework can automatically link system artifacts as an instance of domain classes together in a generated assurance case and there is no mapping relationship needed for a specific project.

Ayoub et al. [18] proposed a safety case pattern for a system developed using a formal method. The pattern considers the satisfaction between a design model in terms of a formal notation and its implementation model. Our safety2design pattern can complement their pattern since both patterns target different phases of a software development process. More important, while Ayoub did not present the details of generating a specific assurance case based on their template, generation of the safety2design is totally built on the model transformation technique, which fits well with the purpose of MDA.

Hauge et al. [19] proposed a pattern-based method to facilitate software design for safety critical systems. Under the pattern-based method is a language that offers six different kinds of basic patterns as well as operators for composition. One of the important ramifications of this method is the generation of safety case, which is connected to the artifacts produced by the same method during a development process.

Jee et al. [7] discussed the construction of an assurance case for the pace-maker software using a model-driven development approach, which is similar to ours. However, their approach emphasizes on the later stage of a software development such as a timed automata model as a design model and C code as implementation language. The approach considers the application of the results from the UPPAAL tool and measurement based timing analysis as evidence.

7 Conclusion and Future Work

In this paper, we present a novel framework which employs safety patterns to generate an assurance case and further support highlight of affected nodes whose underlying reasoning chain can be damaged during software maintenance. To remove the bias towards using this framework, we used the two case studies developed by two different third-party teams to evaluate the framework. The initial experimental results show that the framework is useful in development of safety critical systems and their evaluation. We will further integrate some confidence calculation models into the framework to support evolution of an assurance case as future work.

Acknowledgement. Lin and Shen are partially supported by the Georgeau Construction Research Institute at Western Michigan University. Li is supported by the National Natural Science Foundation of China (Nos. 61472406 and 61532019).

References

1. National Research Council: Critical Code: Software Producibility for Defense. National Academies Press, Washington, D.C. (2010)
2. Organización Internacional de Normalización, ISO 26262: Road Vehicles: Functional Safety, ISO (2011)

3. US Food and Drug Administration (FDA): Guidance for Industry and FDA Staff-Total Product Life Cycle: Infusion Pump–Premarket Notification [510 (k)] Submissions (2010)
4. European Organisation for the Safety of Air Navigation: Preliminary Safety Case for Airports Surface Surveillance. Eurocontrol (2011)
5. Denney, E., Pai, G.: Automating the assembly of aviation safety cases. IEEE Trans. Reliab. **4** (63), 830–849 (2014)
6. Hawkins, R., Habli, I., Kelly, T., McDermid, J.: Assurance case and prescriptive software safety certification: a comparative study. J. Saf. Sci. **59**(11), 55–71 (2013)
7. Jee, E., Lee, I., Sokolsky, O.: Assurance cases in model-driven development of the pacemaker software. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6416, pp. 343–356. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16561-0_33
8. Denney, E., Pai, G.: Tool support for assurance case development. Autom. Softw. Eng. **25**, 435–499 (2018)
9. Ayoub, A., Kim, B., Lee, I., Sokolsky, O.: A safety case pattern for model-based development approach. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 141–146. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_14
10. Hawkins, R., Habli, I., Kolovos, D., Paige, R., Kelly, T.: Weaving an assurance case from design: a model-based approach. In: IEEE 16th International Symposium on High Assurance Systems Engineering (HASE) (2015)
11. Denney, E.W., Pai, G.J.: Safety case patterns: theory and applications (2015)
12. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Proceedings of Future of Software Engineering 2007 (2007)
13. Goal Structuring Notation Working Group: GSN Community Standard Version 1, pp. 437–451 (2011)
14. Adelard (2003). http://adelard.co.uk/software/asce/
15. http://www.eclipse.org/atl/
16. Gross, K.H., Fifarek, A.W., Hoffman, J.A.: Incremental formal methods based design approach demonstrated on a coupled tanks control system. In: 2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE) (2016)
17. Langari, A., Maibaum, T.: Safety cases: a review of challenges (2013)
18. Ayoub, A., Kim, B., Lee, I., Sokolsky, O.: A Systematic approach to justifying sufficient confidence in software safety arguments. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 305–316. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33678-2_26
19. Hauge, A.A., Stølen, K.: A pattern-based method for safe control systems exemplified within nuclear power production. In: Ortmeier, F., Daniel, P. (eds.) SAFECOMP 2012. LNCS, vol. 7612, pp. 13–24. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33678-2_2

Refinement

Correct-by-Construction Implementation of Runtime Monitors Using Stepwise Refinement

Teng Zhang[1](✉), John Wiegley[2], Theophilos Giannakopoulos[2], Gregory Eakman[2], Clément Pit-Claudel[3], Insup Lee[1], and Oleg Sokolsky[1]

[1] University of Pennsylvania, Philadelphia, PA 19104, USA
{tengz,lee,sokolsky}@cis.upenn.edu
[2] BAE Systems, Burlington, MA 01803, USA
{john.wiegley,theo.giannakopoulos,gregory.eakman}@baesystems.com
[3] MIT CSAIL, Cambridge, MA 02139, USA
cpitcla@csail.mit.edu

Abstract. Runtime verification (RV) is a lightweight technique for verifying traces of computer systems. One challenge in applying RV is to guarantee that the implementation of a runtime monitor correctly detects and signals unexpected events. In this paper, we present a method for deriving correct-by-construction implementations of runtime monitors from high-level specifications using Fiat, a Coq library for stepwise refinement. SMEDL (Scenario-based Meta-Event Definition Language), a domain specific language for event-driven RV, is chosen as the specification language. We propose an operational semantics for SMEDL suitable to be used in Fiat to describe the behavior of a monitor in a relational way. Then, by utilizing Fiat's refinement calculus, we transform a declarative monitor specification into an executable runtime monitor with a proof that the behavior of the implementation is strictly a subset of that provided by the specification. Moreover, we define a predicate on the syntax structure of a monitor definition to ensure termination and determinism. Most of the proof work required to generate monitor code has been automated.

Keywords: Runtime monitor · SMEDL · Formal semantics · Coq Stepwise refinement

1 Introduction

Runtime verification (RV) [1] is a lightweight technique for correctness monitoring of critical systems. The objective of RV is to check if a run of the system (referred as a target system in the remainder of the paper), usually abstracted

This work is supported in part by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract FA8750-16-C-0007 and by ONR SBIR contract N00014-15-C-0126.

as a trace of events either from the execution or the logging information, satisfies or violates certain properties. Properties to be checked using RV are usually specified by high level languages, such as temporal logics or state machines. Specifications are then converted into executable monitor code by either a code generator or manual effort. However, informal code generation processes are usually error-prone, and the generated monitor code may not adhere to its specification. During execution, an incorrect monitor may not detect property violations, which can lead to serious consequences in safety critical systems. As a result, it is desirable to use a formal procedure to achieve correct-by-construction implementation of monitors.

This paper presents a method for generating correct-by-construction implementations of runtime monitors written in SMEDL [2], a state-machine-based DSL (domain specific language) for RV. Our method is based on Fiat [3], a powerful deductive synthesis framework embedded in the Coq proof assistant [4]. The core idea in Fiat is to separate declarative specifications from concrete implementations. Users start by embedding their DSL into Coq, so that each DSL program is understood as a mathematical description of the set of results it may return. Using stepwise refinement, each program can be translated into a correct-by-construction executable implementation.

High-level specifications of monitors are attractive because they succinctly describe what monitors should do: implementation details are crucial for performance, but they can be determined separately, along with a proof of preservation of semantics between the specification and implementation. One challenge is to design semantic rules that can be used smoothly in the specification, while remaining amenable to refinement so that such preservation can be proved without excess difficulty.

Additionally, to generate a correct monitor, we have to ensure that the monitor specification is well-formed. In this paper, we require that each monitor satisfies two properties: *termination* and *determinism*. Termination is important because if a monitor goes into infinite loop, or gets stuck during the execution, it will not be able to receive events from the system and catch property violations. Determinism ensures that the monitor always produces the same output given the same input and current state. The code generation process thus needs to be able to detect and reject any "bad" monitor specification that may get stuck during execution.

To overcome the challenges above, we provide a solution for constructing runtime monitor implementation using the Fiat framework. The contributions of this paper are the following:

– We present an operational semantics for SMEDL monitors written in a relational way that ensures that the functionality of the specification and the implementation are separated. This lays a foundation for generating correct code using Fiat.

- We define a predicate on the definition of SMEDL monitors to ensure termination and determinism. Only a well-formed monitor can be extracted into executable code through the Fiat framework.
- We implement a complete procedure from the design of specifications to the generation of correct-by-construction runtime monitors using Fiat, illustrating how deductive synthesis can be used to formally derive trustworthy monitors.

The code generation process presented in this paper is shown in Fig. 1. First, we define a general declarative specification to describe the behavior of a monitor reacting to input events. This specification is independent of specific monitor definitions. It is then refined into a general function using Fiat. To derive executable code for a specific monitor, the user needs to specialize this general function using a definition of that monitor along with a proof of its well-formedness. A Haskell program is extracted and rendered into a monitor to check properties of the target system. Since most of the proof work is automated by auxiliary decision procedures and tactics, applying this methodology does not lead to a heavy workload. Furthermore, efficient monitor code can be generated simply by picking other implementations, which are guaranteed by Fiat to adhere to the behavior described by the specification.

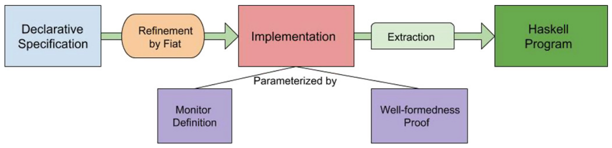


Fig. 1. Code generation process

The paper is organized as follows. Section 2 introduces basic concepts of the Fiat framework and SMEDL. Sections 3 and 4, respectively present the operational semantics of SMEDL and the monitor definition predicates that ensure termination and determinism. Sections 5 and 6 present the process of refinement using Fiat and illustrate the usability of this method by a case study. Section 7 summarizes related work. Section 8 concludes the paper and presents the future work. The code for this paper can be downloaded from the internet[1].

2 Preliminaries

Overview of Fiat. Stepwise refinement derives executable programs from nondeterministic specifications. In each step, some details of the computation are decided upon, proceeding this way until a computable program is derived. Each

[1] https://gitlab.precise.seas.upenn.edu/tengz/smedl-fiat-code.

refinement step must not introduce new behavior: the values that a refined program may produce must be a subset of the values allowed by the specification. Fiat is a stepwise refinement framework, providing a semi-automatic way of deriving correct and efficient programs. Here "semi" means that while the derivation process is automatic, it depends on manually verified refinement lemmas, specific to the domain that Fiat is applied to. This section briefly gives an overview of Fiat. Readers can refer to [3,5,6] for more information.

Important Syntax Structures in Fiat. In Fiat, specifications are logical predicates characterizing allowable output values. These specifications are called computations, and written in the non-determinism monad: deterministic programs can be lifted into computations using the "ret" combinator, computations can be sequenced using the "bind" combinator (written "$x \leftarrow c_1; c_2(x)$"), and a nondeterministic choice operator written $\{a|P\ a\}$ is used to describe programs that may return any value satisfying a logical predicate P. Concretely, the result of binding two computations c_1 and c_2 as shown above is simply the set $\{y|\exists x, x \in c_1 \wedge y \in c_2(x)\}$.

Fiat computations are organized into an Abstract Data Type (ADT), a structure used to encapsulate a set of operations on an internal data type. In Fiat, an ADT contains an internal representation type (denoted as *rep*), a list of constructors for creating values of type *rep*, and a list of methods operating on values of type *rep*. A well-typed ADT guarantees that *rep* is opaque to client programs using the operations of the ADT.

Refinement Calculus in Fiat. Refinement in Fiat is the process of transforming an ADT into a more deterministic ADT, involving refining all constructors and methods defined in ADT and picking an efficient internal representation using data refinement [7] of *rep*. When refining an expression, a partial relation $c_1 \supseteq c_2$ must be preserved for each refinement step, meaning that the possible values of expression c_2 must be a subset of the possible values of expression c_1. For the data refinement part, changes of internal representation are justified using a user-selected abstraction relation, so that if the internal states of two ADTs are related, calling their methods must preserve the relation and produce the same client-visible outputs. Adding the abstraction relation r to the partial relation $\supseteq$ of refinement on expression, Fiat uses $\supseteq_r$ to represent the relation to be preserved for each refinement step: $I_1 \supseteq_r I_2 \supseteq_r ... \supseteq_r I_i$ where I_1 is the initial ADT and I_i is a *fully refined* (i.e. deterministic) ADT.

SMEDL Concepts. A SMEDL monitor is a collection of *scenarios*. Each scenario is an Extended Finite State Machine (EFSM) [8] in which the transitions are performed by reacting to events. By specifying a set of scenarios in a SMEDL monitor, the monitor can check the behavior of a certain aspect of the system from a variant levels of abstraction with a clearer separation. Scenarios interact with each other using shared state variables or by triggering execution of other scenarios through raised events. There are three types of events: *imported*, *exported* and *internal*. Imported events, which are responsible for triggering the execution of a monitor, are raised from the target system or by other monitors; exported events are raised within the monitor and are then sent to other monitors; internal events are used to trigger transitions, but are only seen and processed within a given monitor. Each transition is labeled with a triggering event and attached to a guard condition and a list of actions to be executed after the transition. Primitive data types, such as arithmetic and logical operations, are supported in SMEDL. The abstract syntax of SMEDL is given below.

A monitor is a 3-tuple $\langle V, \Sigma, S\rangle$, where V is a set of state variables, Σ is a set of event declarations and S is a non-empty set of scenarios for the monitor. The event declaration is a three-tuple $\langle \mathit{eventType}, \mathit{eventName}, \mathit{attributeTypes}\rangle$ where *eventType* is an enumeration over three types of event introduced above; *attributeTypes* represents the list of types of attributes of the event.

A scenario is a 5-tuple $\langle n, Q, q_0, E, \delta\rangle$ where n, Q, and q_0 are respectively the identity, the set of states and the initial state, E is the set of *alphabets*—events that can trigger the transitions of the machine and δ is the set of transitions of the scenario.

A transition is a four-tuple $\langle \mathit{qsrc}, \mathit{qdst}, \mathit{stpEv}, \mathcal{A}\rangle$ where *qsrc* and *qdst* are the source and target state of the transition, *stpEv* denotes the triggering event of the transition, typed with *eventInstance* which is a three-tuple $\langle \mathit{event}, \mathit{eventArgs}, \mathit{eventGuard}\rangle$. *event* is a reference to the corresponding event declaration, *eventArgs* is defined as the list of local variable names, and *eventGuard* is an expression guarding the transition. Both the state variables and local variables can be used in the *eventGuard*. The set $\mathcal{A}$ consists of statements to be executed immediately after the transition, which can either update state variables or raise events. Note that we assume all transitions in the scenario are complete.

3 An Operational Semantics of SMEDL

This section proposes the formal semantic rules for a single monitor. When an imported event is sent to a monitor, state transitions within the monitor are triggered. Actions attached to transitions can raise internal or exported events. Internal events are used to trigger further transitions in other scenarios. After all triggered transitions are completed, exported events are output and the monitor waits for the next imported event. This process is denoted as a *macro-step*, which cannot be interrupted by other imported events. Each scenario can execute its transition at most once in a single macro-step so that there is no infinite loop of interaction between scenarios. We introduce a data structure *configuration* to describe the dynamic state of a monitor.

Configurations. A configuration is a five-tuple $\langle \mathit{MS}, \mathit{DS}, \mathit{PD}, \mathit{EX}, \mathit{SC}\rangle$. *MS* denotes the mapping from scenarios to their current states; *DS* is a well-typed mapping from state variables to values; *PD* is the set of pending events to trigger transitions within the monitor; and *EX* is the set of raised, exported events. Elements in both sets are events binding with actual attribute values, denoted as *raisedEvents*. *SC* is the set of scenarios executed during the current macro-step and its corresponding triggering events. Each configuration *conf* relates to a monitor M, denoted as $conf_M$. The subscript is omitted in the remainder of the paper whenever the context is clear.

A macro-step is constructed by chaining a series of consecutive *micro-steps*. Each micro-step is the synchronous composition of a set of transitions on scenarios with the same triggering event, constructed by the interleaved application of the *basic rule* and the *synchrony rule*. The chaining of micro-steps is performed by applying the *chain merge rule*.

Basic Rule. The basic rule is applied to a state machine whenever a transition is triggered by a pending event. In the definition below, the scenario performing the transition is *mh*, *conf* denotes the configuration before applying the rule, and $conf'_{mh}$ denotes the configuration after applying the rule, on *mh*.

$$\frac{\begin{array}{c} tr : s1 \xrightarrow{e\{a\}}_{c} s2 \\ valid(tr, conf, mh) \\ conf'_{mh} = updateConfig(mh, conf, tr) \end{array}}{conf \xrightarrow{e} conf'_{mh}}$$

tr is the enabled transition from *s1* to *s2* by *e*; *a* is the set of actions for *tr*; and *c* is the guard. *valid* tests the validity of *tr* under the configuration *conf*, which includes: (1) *tr* is the transition of *mh*, (2) current state of *mh* is *s1* and (3) *c* evaluates to true for current *DS* and attribute values of the event, (4) *e* is in *PD*, and (5) *mh* is not in *SC*. When *tr* is taken, *conf* is updated by executing the function *updateConfig*, denoted as $conf \xrightarrow{e} conf'_{mh}$. The update includes: (1) *mh* transitions to *s2* and is put into *SC*, (2) *DS* is updated by the actions in the transition, (3) *e* is removed from *PD*, and (4) raised events are respectively added to *PD* and *EX* according their types.

Synchrony Rule. One or more scenarios are enabled by a triggering event from a source configuration. The basic rule creates new configurations for each scenario by taking these transitions. The synchrony rule then combines scenario's resulting configuration into a new configuration. Combination of two configurations *conf₁* and *conf₂* under the origin configuration *conf* is defined below.

– $\forall mh \in S$,

$$MS_{conf'}(mh) = \begin{cases} MS_{conf_1}(mh) \;, MS_{conf_1}(mh) = MS_{conf_2}(mh) \\ MS_{conf_1}(mh) \;, MS_{conf_1}(mh) \neq MS_{conf}(mh) \\ MS_{conf_2}(mh) \;, MS_{conf_2}(mh) \neq MS_{conf}(mh) \end{cases}$$

– $\forall v \in V$,

$$DS_{conf'}(v) = \begin{cases} DS_{conf_1}(v), \;\; DS_{conf_1}(v) = DS_{conf_2}(v) \\ DS_{conf_1}(v), \;\; DS_{conf_1}(v) \neq DS_{conf}(v) \\ DS_{conf_2}(v), \;\; DS_{conf_2}(v) \neq DS_{conf}(v) \end{cases}$$

– $PD_{conf'} = PD_{conf_1} \cup PD_{conf_2}$
– $EX_{conf'} = EX_{conf_1} \cup EX_{conf_2}$

– $SC_{conf'} = SC_{conf_1} \cup SC_{conf_2}$

The synchrony rule is given below. *confs* is the set of target configurations obtained from the basic rule given a source configuration *conf* and an event *e*. *MergeAll* combines each configuration in *confs* into a new configuration by repeatedly combining configurations pairwise. The micro-step from c to c' by e is denoted as $c \overset{e}{\hookrightarrow} c'$.

$$\frac{confs = \{conf_{mh} | conf \xrightarrow{e} conf_{mh}\}}{conf \overset{e}{\hookrightarrow} MergeAll(confs)}$$

Chain Merge Rule. The objective of the chain merge rule is to construct a macro-step, defined inductively below.

$$\frac{conf \overset{e1}{\hookrightarrow} conf'}{conf \overset{e1}{\rightharpoondown}_1 conf'} e1 \in ImportedEvents \tag{1}$$

$$\frac{\begin{array}{c} conf \overset{e1}{\rightharpoondown}_n conf' \\ conf' \overset{e2}{\hookrightarrow} conf'' \end{array}}{conf \overset{e1}{\rightharpoondown}_{n+1} conf''} \tag{2}$$

Case (1) shows that a micro-step triggered by an imported event is the basic case. The corresponding source configuration is denoted as an *initial* configuration in the remainder of this paper. The inductive case is shown in case (2). Note that there is no restriction on how to choose $e2$ from *PD* of *conf'*. The subscript in the chain merge rule indicates the number of micro-steps from the initial configuration to the current configuration.

Discussion on Design of Semantic Rules. The basic rule and synchrony rule are encoded in Coq as functions, because the transition of a scenario and the construction of a micro-step are deterministic. On the other hand, the chain merge rule is defined relationally because it does not specify which event to choose from *PD* to trigger the next micro-step, nor does it guarantee termination during the combination of micro-steps. To derive a computable version, which must terminate because of restrictions in Coq, we require predicates, given below, on the syntactic structure of monitor specifications, such that termination and determinism are guaranteed.

4 Towards a Well-Formed Monitor Specification

This section presents the definition of a *well-formed* monitor. Both the basic and synchrony rule are partial so we need to make sure that their application succeeds. Moreover, two vital properties for a monitor, termination and determinism, are considered. A set of predicates are proposed from which the definition of well-formedness is constructed. We prove that if a monitor satisfies these predicates, it always terminates in a final state, deterministically, which indicates that the monitor is well-formed, i.e. no runtime errors are possible. These predicates are required since only well-formed monitor specifications may be generated into executable code by Fiat.

4.1 Well-Formedness Predicates

Table 1 lists predicates for well-formedness, which are divided into three categories indicating which part of the execution is influenced by the predicates.

Table 1. Predicates for well-formedness

Classification	Name	Definition
Scenario level	P1	$\forall s \in S_M,\ \forall tr_1\ tr_2 \in \delta_s, qsrc_{tr_1} = qsrc_{tr_2}\ \wedge\ event_{stpEv_{tr_1}} = event_{stpEv_{tr_2}} \Rightarrow eventGuard_{stpEv_{tr_1}} = \neg eventGuard_{stpEv_{tr_2}}$
	P2	$\forall s \in S_M,\ \forall tr \in \delta_s,\ \forall e \in \Sigma_M,\ event_{stpEv_{tr}} = e\ \Rightarrow\ e \in E_s$
Micro-step level	P3	$\forall v \in V_M\ , \forall sce1\ \ sce2 \in S_M\ ,\ updateVar(v, sce1)\ \wedge\ updateVar(v, sce2) \Rightarrow E_{sce1} \cap E_{sce2} = \emptyset$
Macro-step level	P4	$\forall e \in \Sigma_M, eventType_e = Imported\ \vee\ eventType_e = Internal \Rightarrow \exists sce, sce \in S_M\ \wedge\ e \in E_{sce}$
	P5	$\forall e\ e1\ e2 \in \Sigma_M, e1 \neq e2 \wedge e \Uparrow_M e1 \wedge e \Uparrow_M e2 \Rightarrow \neg\exists sce, sce \in S_M\ \wedge\ e1 \in E_{sce}\ \wedge\ e2 \in E_{sce}$
	P6	$\forall e\ e1 \in \Sigma_M, eventType_e = Imported\ \wedge\ e \neq e1\ \wedge\ e \Uparrow_M e1 \Rightarrow \neg\exists sce, sce \in S_M\ \wedge\ e \in E_{sce}\ \wedge\ e1 \in E_{sce}$
	P7	$\forall e \in \Sigma_M, \forall\ sce1\ \ sce2 \in S_M, raiseEv(sce1, e)\ \wedge\ raiseEv(sce2, e)\ \wedge\ sce1 \neq sce2\ \Rightarrow \neg\exists e' \in \Sigma_M, triggerSce(sce1, e')\ \wedge\ triggerSce(sce2, e')$
	P8	$\forall e \in \Sigma_M,\ sce \in S_M,\ stp \in \delta_{sce}, noDuplicatedRaise(e, sce, stp)$
	P9	$\forall e1\ e2 \in \Sigma_M,\ \forall v \in V_M, \exists e \in \Sigma_M, noDependency(e, e1, e2) \wedge updateVarEv(v,e1) \Rightarrow \neg updateVarEv(v,e2)\ \wedge \neg usedVarEv(v,e2)$

P1 and *P2* guarantee that exactly one transition is triggered for a scenario during the application of the basic rule, by an event from the alphabet for that scenario. *P3* guarantees that when applying the synchrony rule to construct a micro-step, scenarios that share the same triggering event never update the same variable. $updateVar(v, sce)$ means that variable v is updated by actions from the transitions of scenario sce.

A well-formed monitor guarantees that it always terminates in some final state. The definition of a final configuration is given below:

Definition 1 (Final Configuration). *A configuration conf is a final configuration if (1) $SC_{conf} \neq \emptyset$ and (2) $PD_{conf} = \emptyset$.*

The tricky part is that all pending events must be consumed at the end of each macro-step, i.e. there are no pending events when all the available scenarios have finished execution and that the execution of a monitor never gets stuck because of a mismatch between enabled scenarios and pending events.

P4 guarantees that all imported events or internal events can trigger execution of some scenarios. *P5* and *P6* ensure that imported or internal events that may be raised in the same macro-step cannot directly trigger execution of the same scenario. $e \Uparrow_M e1$ means that $e1$ is raised by the actions of transitions transitively triggered by e. *P7* and *P8* guarantee that in each macro-step, an internal event cannot be raised multiple times. *raiseEv(sce,e)* means that the actions of transitions defined in *sce* contain raising *e*. *triggerSce(sce,e)* means that *e* may transitively trigger transition of *sce*. *noDuplicatedRaise(e,sce,stp)* means that *e* can only be raised once in *stp* of *sce*.

The chain merge rule does not specify an order for the chaining of micro-steps. If a monitor is not well defined, the execution result of a macro-step could be non-deterministic. This is undesirable because we always want a deterministic verdict from a monitor given the same input. *P1* and *P2* ensure scenario-level determinism. *P5* to *P8* also prevent some behaviors that may lead to non-determinism. We define a proposition *noDependency(e,e1,e2)* $\overset{\text{def}}{=}$ $eventType_e = Imported \wedge e \Uparrow_M e1 \wedge e \Uparrow_M e2 \wedge \neg e1 \Uparrow_M^e e2 \wedge \neg e2 \Uparrow_M^e e1$. This means that $e1$ and $e2$ may be raised in the macro-step triggered by imported event e, and that during this macro-step, $e1$ can not transitively raise $e2$, and vice versa. *P9* guarantees that updating a state variable is mutually exclusive. *updateVarEv(v,e)* and *usedVarEv(v,e)* respectively mean that v cannot be updated and used in any actions transitively triggered by e.

We use the notation $Pi(M)$ to represent that a monitor M satisfies predicate Pi. A well-formed monitor satisfies the nine predicates defined above, *Wellformed(M)* $\overset{\text{def}}{=} \bigwedge_{1 \leq i \leq 9} Pi(M)$.

4.2 Proof of Termination and Determinism

Given a monitor that is well-formed, and which starts execution with a well-typed imported event, we can now prove that it can always reach a final state within a limited number of micro-steps, as described in Theorem 1 below:

Theorem 1 (Termination). *Given a well-formed monitor M, two of its configurations $conf_M$ and $conf'_M$ and an imported event e, if $conf_M \xrightarrow{e}_n conf'_M$ and M cannot take any micro-step from $conf'_M$, $conf'_M$ is a final configuration and n is equal to or less than $|S_M|$.*

To prove this theorem, we need to first prove that the number of micro-steps taken within a macro-step is bounded. Because each scenario can only transition once during each macro-step, and at least one scenario executes in each micro-step, the number of micro-steps to be taken is bounded by the number of scenarios of the monitor. So we first prove that $|SC_{conf}|$ strictly increases in a micro-step.

Lemma 1 (Increase of SC). *Given two configurations conf conf′ and an event e, if* $conf \overset{e}{\hookrightarrow} conf'$*, then* $|SC_{conf}| < |SC_{conf'}|$.

With Lemma 1 and the fact that SC_{conf_M} is a subset of $|S_M|$, we can prove that the number of micro-steps taken by a well-formed monitor in a macro-step is bounded by the number of scenarios:

Lemma 2 (Up-bound of micro-steps). *Given a well-formed monitor M, two of its configurations* $conf_M$ *and* $conf'_M$ *and an imported event e, if* $conf_M \overset{e}{\rightharpoonup}_n conf'_M$*, then* $n \leq |S_M|$.

Next we need to prove that macro-step has the *progress* property, which guarantees that a well-formed monitor cannot be stuck in a non-final state:

Lemma 3 (Progress). *Given a well-formed monitor M, two of its configurations* $conf_M$ *and* $conf'_M$ *and an imported event e, if* $conf_M \overset{e}{\rightharpoonup}_n conf'_M$ *and* $conf'_M$ *is not a final configuration, then M can take a micro-step on all of its pending events from* $conf'_M$.

With the three core lemmas presented above, and other auxiliary lemmas, Theorem 1 can be proved. With this theorem, we can always pick a terminating implementation of the relational semantic rules during the refinement step.

Deterministic execution of a macro-step is represented by the theorem below:

Theorem 2 (Determinism). *Given a well-formed monitor M, if* $conf_M \overset{e}{\rightharpoonup} conf'_M$, $conf_M \overset{e}{\rightharpoonup} conf''_M$ *and both* $conf'_M$ *and* $conf''_M$ *are final configurations, then* $conf'_M = conf''_M$.

This theorem is proved using the idea of Newman's lemma [9]. First, we prove the *diamond* lemma defined below:

Lemma 4 (Diamond). *Given a well-formed monitor M, if* $conf_M$ *is an initial configuration or there exists a configuration oconf such that* $oconf \overset{e}{\rightharpoonup} conf_M$*, and* $conf_M \overset{e1}{\hookrightarrow} conf1_M$ *and* $conf_M \overset{e2}{\hookrightarrow} conf2_M$*, then there exists a configuration* $conf'_M$ *such that* $conf1_M \overset{e2}{\hookrightarrow} conf'_M$ *and* $conf2_M \overset{e1}{\hookrightarrow} conf'_M$.

Then, by induction on the number of micro-steps to be taken by two transition chains, we can prove the *confluence* lemma:

Lemma 5 (Confluence). *Given a well-formed monitor M, if* $conf_M \overset{e}{\rightharpoonup} conf1_M$, $conf_M \overset{e}{\rightharpoonup} conf2_M$*, there exists a configuration* $conf'_M$ *such that* $conf1_M \hookrightarrow_* conf'_M$ *and* $conf2_M \hookrightarrow_* conf'_M$.

Transition $\hookrightarrow_*$ represents multiple micro-steps. Lemma 5 ensures that if an initial configuration *conf* can transition into two non-final configurations $conf_1$ and $conf_2$, then they can always transition back to the same configuration. Using Lemma 5 and the fact that a final configuration cannot take any micro-step, Theorem 2 can be proved.

5 Refinement of a Monitor Specification Using Fiat

This section presents how to generate correct-by-construction code from a declarative ADT using Fiat. Figure 2 gives an overview of the code generation process. The initial ADT describes the basic behavior of monitors in a declarative way using semantic rules defined in the previous section. Then, the ADT is refined by proving a "sharpening" theorem, wherein the representation type, constructors and methods of the ADT are refined. The refinement of methods involves picking a specific implementation and proving that $\supseteq_r$, introduced in Sect. 2, is preserved between the specification and the implementation. The implementation is parameterized by a specific monitor definition given a starting state and proof of well-formedness of that monitor. Haskell code can then be extracted from this definition.

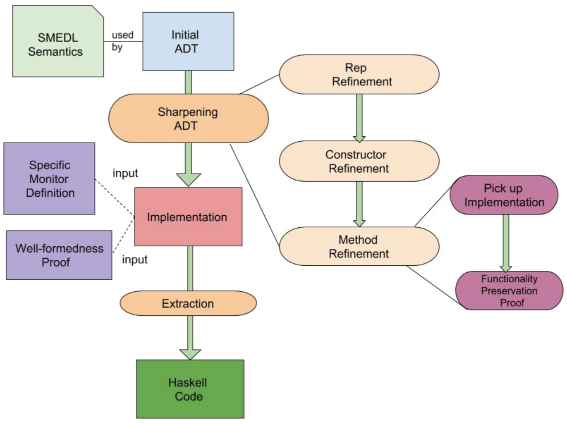


Fig. 2. Fiat refinement steps for code generation

5.1 Definition of an ADT

Basically, the monitor ADT describes the common process of handling imported events using the semantic rules defined in the previous section. The definition of this ADT is given below.

```
Definition confSpec : ADT _ := Def ADT {
  rep := configuration M,
  Def Constructor0 newS : rep := { c : configuration M | readyConfig c },,
  Def Method1 processEventS (r : rep) (e: raisedEvent | raisedAsImported  M e) :
      rep * list raisedEvent :=
      { p : rep * list raisedEvent
    | exists conf' econf,
        chainMergeTrans r conf' econf (' e) (fst p) (snd p) }
}.
```

The configuration of a given monitor M is used as the representation type for the ADT. Instead of constructing a concrete value, Constructor *newS* specifies that the starting state of a monitor should be a *ready* configuration. A ready configuration has empty sets for *PD*, *EX* and *SC*, indicating that the monitor is ready to receive an imported event for the next macro-step. The method *processEventS* specifies the non-deterministic action of taking a macro-step. The first parameter r represents the current ready state of the monitor and the second parameter e is the imported event triggering the macro-step. The return value is a tuple of a ready configuration that reflects the updated state of the monitor after the macro-step and a list of raised exported events. The semantic rules from previous sections were defined in a relational way to conveniently specify this method, since relations easily model non-deterministic functions. To adapt the chain merge rule to the interface of *processEventS*. *chainMergeTrans* is defined below:

```
Definition chainMergeTrans {M : monitor} (conf conf' econf: configuration M)
(e: raisedEvent) (rconf: configuration M)  (events: list raisedEvent)  : Prop :=
   configTrans conf conf' /\
   chainMerge conf' econf e /\
   finalConfig econf  /\
   configTransRev econf rconf /\
   events = EX econf.
```

configTrans conf conf' represents the transformation from ready configuration *conf* to initial configuration *conf'*; *chainMerge conf' econf e* is the Coq definition of $conf' \xrightarrow{e} econf$ with the number of steps taken omitted; and *configTransRev* represents the transformation from *econf* to a new ready configuration *rconf*. *events* is the set of exported events raised in this macro-step.

5.2 Refinement Process

Refinement by Fiat requires proving the theorem *FullySharpened(confSpec M)*, parameterized over some monitor definition M. The implementation is wrapped in the proof term of the theorem. The first step refines the representation type. In this paper, we choose the same representation type—the configuration of monitor M—in the implementation. As a result, the *abstraction relation r* is plain equality. Constructor *newS* is refined by choosing a ready configuration *conf* for monitor M, given by the starting state of monitor M. Just like parameter M, *conf* also needs to be provided to generate a concrete, executable monitor. To refine method *processEventS*, we need to provide a deterministic function that preserves the semantics of applying the chain merge rule. Preservation of the specification's semantics for this function is given by the lemma below:

```
Lemma ProcessEventRefined M (C : configuration M) (W : Wellformed M)
      (Cor:readyConfig C) (e: raisedEvent) (P : raisedAsImported M e) :
 refine { p : configuration M * list raisedEvent
        | exists conf' econf, chainMergeTrans C conf' econf e (fst p) (snd p) }
        (ret (macroStepReadyFinal W Cor P
                                   (length (S M)))).
```

macroStepReadyFinal is a function which takes a ready configuration C and returns a new ready configuration and list of exported events. In this paper, we choose a straightforward implementation: a fixpoint function that picks the first event from PD of the current configuration to trigger the next micro-step. Note that in the Coq definition, we use a list to represent the set, and due to the predicates establishing well-formedness, PD can never have duplicate events. The number of times the semantic function gets invoked is bounded by the number of scenarios in M. Provided that M is well-formed, it is guaranteed that the resulting configuration is a final configuration. The lemma *ProcessEventRefined* establishes that the return value is a subset of the results obtained by applying *chainMergeTrans* used in the original ADT. From the proof term of the theorem, an executable version of *processEventS* can be obtained.

It is worth noting that, the semantics of SMEDL can be directly expressed as a Coq function for generating the Haskell code by native Coq. But through Fiat, we can refine from the declarative SMEDL semantics to a more efficient implementation by changing the data structure for configuration, handling pending events more wisely, etc. Moreover, refinement can be conducted in a more mechanical and extensible way in Fiat than using native Coq.

6 Case Study

A general event processing function is generated by refinement, parameterized by: a specific monitor specification, its well-formedness proof and a starting, ready state for that monitor. Therefore, to obtain a correct-by-construction monitor, one needs to (1) write a monitor definition M; (2) prove that M is well-formed; and (3) specify a starting state. A Haskell program may then be extracted, from which a monitor is implemented by adding glue code to receive events from the target system. This section uses a real-world monitoring requirement to illustrate the usability of this method.

SMEDL Specification. The monitoring requirement comes from a known vulnerability(CVE-2017-9228)[2] in Oniguruma v6.2.0 [10], which is related to incorrect parsing of regular expressions, resulting in a crash due to access of an uninitialized variable. Based on a high-level specification, and agnostic of the specific vulnerability, a SMEDL monitor is constructed to detect this violation of the specification and raise an alarm. The specification is based on a part of the regular expression grammar concerning *character classes*. The parsing can be described as the state machine in Fig. 3, where transition labels are omitted for clarity. Transitions in the state machine are triggered by tokens read by the parser and guarded with additional conditions.

Part of the SMEDL specification (denoted as *parseCC*) is given below. To simplify the presentation, we concentrate only on one guard condition, which states that the VALUE state cannot be recursively entered (i.e., from the START state) while a class is still being processed. We refer to this condition below as *in_class* being equal to 1. Scenario *main* records transitions of the parser state machine affected by the code; they are triggered by events that correspond

[2] https://nvd.nist.gov/vuln/detail/CVE-2017-9228.

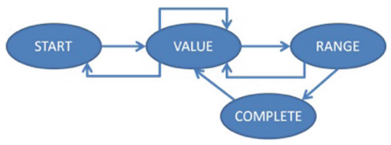


Fig. 3. State machine of parsing character class

by changes to state variables in the code. Scenario *check_class* determines the value of the state variable *in_class* by receiving the imported events *in_class* and *out_class*. The specification can be directly mapped to an AST definition in Coq.

```
object parseCC
    state
        int in_class = 0;
    events
        imported inClass();//enter next_value_class
        imported outClass();//exit next_value_class
        imported state_to_start();//state is set to START
        imported state_to_value();//state is set to VALUE
        imported state_to_range();//state is set to RANGE
        imported state_to_complete();//state is set to COMPLETE
        exported error(int);
    scenarios
       main:
           START -> state_to_value() when (in_class != 1) -> VALUE
           START -> state_to_value() when (in_class == 1)
           {raise error(0);} -> START
           VALUE -> state_to_value() -> VALUE
           VALUE -> state_to_range() -> RANGE
           VALUE -> state_to_start() -> START
           ...

       check_class:
          idle -> in_class() when (in_class == 0)
                {in_class = 1;} -> idle
          idle -> out_class() when (in_class == 1)
                {in_class = 0;} -> idle
```

Proof of Well-Formedness. Proving the well-formedness of a monitor seems hard because there are nine sub predicates needed to be proved and type correctness needs to be checked. However, we have implemented decision procedures to check whether a monitor satisfies *P1* to *P4*. Rest of them can be proved using the auxiliary lemmas and tactics. The LOC for the proof is less than 1k of Gallina and Ltac code. The time for proving well-formedness of *parseCC* is estimated to be about 30 min for a user with basic experience of Coq.

Construction of the Haskell Monitor. The core building block of a *parseCC*-based monitor is given below. *processEventsS* is the general event handling function refined from the Fiat ADT. The Parameter *r* contains the information to be used by *parseCC*: the proof of well-formedness (denoted as *Well_ParseCC*) and a starting state. Parameter *e* is the imported, triggering event for *parseCC*.

```
Definition parseCC_processEvent (r : ComputationalADT.cRep
            program Well_ParseCC configuration1_ready)
           (e: raisedEvent | raisedAsImported  parseCC e) :=
  processEventsS r e.
```

Coq provides the ability to extract Coq definitions to a Haskell program. The monitor is constructed by adding glue code for receiving events from the target system. We compile the Haskell code into an object file and expose two functions to be instrumented into the target program. The type signature of these two Haskell functions are given below:

```
cInitialRep :: IO (Ptr ())
cHandleImported :: CString -> Ptr () -> IO (Ptr ())
```

Both functions rely on the extracted Haskell code. *cInitialRep* provides a starting state for the monitor. *chandleImported* takes the name of an imported event, and the current state of the monitor, and returns a new state with any exported events printed out. The target system is responsible for recording this state update transparently. Using the GHC compiler, both an object file and a C header file are generated. The header file contains the C API of the two functions defined above, which are called in the source code of Oniguruma. When an incorrect transition occurs in the library, an alarm is raised and printed to the screen.

The LOC for the automatically extracted code is about 6k lines but only about 10% of the code depends on the definition of a monitor. For another example monitor with 6 scenarios and 16 transitions, the LOC of the part depending on the monitor definition is less than 1k lines. Thus, scalability would not be an issue.

The difficult part of deriving a monitor is its proof of well-formedness, which can be simplified using the provided decision procedures and tactics. The other steps are easily implemented using common procedures. The methodology presented in this paper provides an straightforward way to implement correct-by-construction monitors.

7 Related Work

We summarize a representative selection of related work in three categories: (1) formal semantics for RV; (2) mechanization of semantics for state-machine-based formalisms and (3) case studies of using the Fiat framework.

The semantics of temporal logic and traces used for RV have been widely studied [11–15]. There has been a lot of work related to describing the semantics of LTL/MTL using automata-like formalisms. Giannakopoulou and Havelund present a technique translating LTL formulae into FSM to monitor program behavior [16]. Drusinsky presents TLCharts, a formalism resembling Harel statecharts while supporting the specification of nondeterministic, temporal properties described in MTL or LTL inside a statechart specification [17]. This semantics is described using Equivalent Non-Deterministic Automaton (ENFA). Roşu and Havelund propose a method for rewriting LTL formulas into binary transition tree finite state machines for online monitoring [18]. Several RV tools support using FSM to specify properties [19–24]. However, little work has been done on mechanizing semantics of DSLs for RV. Our work shows that the semantic mechanization is a necessary foundation for generating correct monitors.

In [25], Paulin-Mohring presents the model of timed automata in Coq for specifying and verifying of telecommunication protocols. Kammüller and Helke [26] formalize the semantics of Statecharts [27] using Isabelle/HOL [28]. In AADL (Architecture Analysis and Design Language) [29], the thread model and mode change are represented using automata. Yang et al. [30] propose a machine-checked transformation of a subset of AADL into TASM (Timed Abstract State Machine [31]). The main purpose of defining formal semantics in these studies is to prove properties of formal models. Although the formal semantics of SMEDL can be used to prove properties of SMEDL or a monitor, the primary objective of our work is code generation. Particularly, the semantic rules have been designed to be conveniently integrated into a Fiat specification.

Correct-by-construction implementation generation using refinement has been well studied [7,32–36]. For instance, Event-B [37] refines an abstract transition system into a more concrete one by adding transitions and states. Fiat is a more general tool, suitable for the refinement objective of SMEDL. Fiat provides flexible support for refinement of libraries for different functionality domains. Delaware et al. [3] illustrate an example of using Fiat to synthesize query structures. Wiegley and Delaware [6] use Fiat to generate efficient and correct implementations of a subset of the bytestring library in Haskell. Chlipala et al. [5] present the development of a simple packet filter in Fiat. The ADT for SMEDL is not as complicated as the case studies listed above, but it is an initial work using Fiat to refine a state-machine-based DSL. The semantic rules and ADT design presented in this paper offer guidance for applying Fiat to generate code for other state-machine-style DSLs.

8 Discussion and Conclusions

We have presented a method for deriving correct-by-construction monitor code using the Fiat framework. An operational semantics of SMEDL is designed using Fiat to describe the essential behavior of any monitor. Using the mechanisms provided by Fiat, this ADT is then refined into executable monitor code while preserving those semantics. We have also proposed a well-formedness predicate on monitor structures and proved that if a monitor is well-formed, it can always terminate deterministically in a final state after reacting to any imported event.

One concern of using formal techniques is the manual effort involved in proof work. In our development, proofs are divided into two parts: One part includes proofs used during the refinement process, and auxiliary tactics and decision procedures for proving the well-formedness of any monitor; the other part is the proof of well-formedness for a particular monitor. The raw LOC in Coq for the first part is about 30k lines. However, to apply the technique, users only need prove well-formedness of their particular monitor, which is not labor-intensive given the help of auxiliary tactics and lemmas. Therefore, we assert that generating correct runtime monitors using a proof assistant is a feasible task.

One main avenue of future work would be to improve the definition of well-formedness, and implement more auxiliary tactics for better usability. It is also worth exploring more efficient implementation of the semantic rules in order to generate more optimized monitor code.

References

1. Sokolsky, O., Havelund, K., Lee, I.: Introduction to the special section on runtime verification. Softw. Tools Technol. Transf. **14**(3), 243–247 (2012)
2. Zhang, T., Gebhard, P., Sokolsky, O.: SMEDL: combining synchronous and asynchronous monitoring. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 482–490. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_32
3. Delaware, B., Pit-Claudel, C., Gross, J., Chlipala, A.: Fiat: Deductive synthesis of abstract data types in a proof assistant. In: ACM SIGPLAN Notices, vol. 50, pp. 689–700. ACM (2015)
4. The Coq Development Team: The Coq Proof Assistant Reference Manual
5. Chlipala, A., et al.: The end of history? using a proof assistant to replace language design with library design. In: SNAPL 2017: 2nd Summit on Advances in Programming Languages (2017)
6. Wiegley, J., Delaware, B.: Using Coq to write fast and correct Haskell. In: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, pp. 52–62. ACM (2017)
7. Hoare, C., et al.: Data refinement refined (1985)
8. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: Proceedings of the 30th International Design Automation Conference, pp. 86–91. ACM (1993)
9. Newman, M.H.A.: On theories with a combinatorial definition of "equivalence". Ann. Math. **43**, 223–243 (1942)
10. Oniguruma contributors: Oniguruma. https://github.com/kkos/oniguruma. Accessed 27 Mar 2018
11. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24622-0_5
12. Allan, C., et al.: Adding trace matching with free variables to AspectJ. In: ACM SIGPLAN Notices, vol. 40, pp. 345–364. ACM (2005)
13. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. J. Log. Comput. **20**(3), 675–706 (2010)
14. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_23
15. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: Departmental Papers (CIS), pp. 294 (1999)
16. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: 2001 Proceedings of 16th Annual International Conference on Automated Software Engineering. (ASE 2001), pp. 412–416. IEEE (2001)
17. Drusinsky, D.: Semantics and runtime monitoring of tlcharts: statechart automata with temporal logic conditioned transitions. Electron. Notes Theor. Comput. Sci. **113**, 3–21 (2005)

18. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. **12**(2), 151–197 (2005)
19. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03240-0_13
20. Havelund, K.: Runtime verification of C programs. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) FATES/TestCom -2008. LNCS, vol. 5047, pp. 7–22. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68524-1_3
21. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Transf. **14**(3), 249–289 (2012)
22. Luo, Q., et al.: RV-monitor: efficient parametric runtime verification with simultaneous properties. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 285–300. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_24
23. Chen, Z., Wang, Z., Zhu, Y., Xi, H., Yang, Z.: Parametric runtime verification of C programs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 299–315. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_17
24. Reger, G., Cruz, H.C., Rydeheard, D.: MarQ: monitoring at runtime with QEA. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 596–610. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_55
25. Paulin-Mohring, C.: Modelisation of timed automata in Coq. In: Kobayashi, N., Pierce, B.C. (eds.) TACS 2001. LNCS, vol. 2215, pp. 298–315. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45500-0_15
26. Kammüller, F., Helke, S.: Mechanical analysis of UML state machines and class diagrams. In: The Proceedings of Workshop on Precise Semantics for the UML. ECOOP2000. Citeseer (2000)
27. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
28. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
29. Frana, R., Bodeveix, J.P., Filali, M., Rolland, J.F.: The AADL behaviour annex-experiments and roadmap. In: 2007 12th IEEE International Conference on Engineering Complex Computer Systems, 377–382. IEEE (2007)
30. Yang, Z., Hu, K., Ma, D., Bodeveix, J.P., Pi, L., Talpin, J.P.: From AADL to timed abstract state machines: a verified model transformation. J. Syst. Softw. **93**, 42–68 (2014)
31. Ouimet, M., Lundqvist, K., Nolin, M.: The timed abstract state machine language: an executable specification language for reactive real-time systems. In: RTNS 2007, p. 15 (2007)
32. Dijkstra, E.W.: A constructive approach to the problem of program correctness. BIT Numer. Math. **8**(3), 174–186 (1968)
33. Srinivas, Y.V., Jüllig, R.: Specware: formal support for composing software. In: Möller, B. (ed.) MPC 1995. LNCS, vol. 947, pp. 399–422. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60117-1_22

34. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_12
35. Cohen, C., Dénès, M., Mörtberg, A.: Refinements for free!. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 147–162. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_10
36. Lammich, P.: Refinement to imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_17
37. Abrial, J.R., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: application to event-B. Fundam. Inform. **77**, 1–28 (2007)

Identifying Microservices Using Functional Decomposition

Shmuel Tyszberowicz[1], Robert Heinrich[2], Bo Liu[3,4](✉), and Zhiming Liu[3]

[1] The Academic College Tel-Aviv Yafo, Tel Aviv, Israel
tyshbe@mta.ac.il
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
heinrich@kit.edu
[3] Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing, China
[4] Southwest University, Chongqing, China
{liubocq,zhimingliu88}@swu.edu.cn

Abstract. The microservices architectural style is rising fast, and many companies use this style to structure their systems. A big challenge in designing this architecture is to find an appropriate partition of the system into microservices. Microservices are usually designed intuitively, based on the experience of the designers. We describe a systematic approach to identify microservices in early design phase which is based on the specification of the system's functional requirements and that uses functional decomposition. To evaluate our approach, we have compared microservices implementations by three independent teams to the decomposition provided by our approach. The evaluation results show that our decomposition is comparable to manual design, yet within a much shorter time frame.

Keywords: Microservices · Decomposition · Coupling · Cohesion Clustering

1 Introduction

The microservices architecture style is rising fast as it has many advantages over other architectural styles such as scalability (fine-grained, independently scalable), improved fault isolation (and thus resilience), and enhanced performance. Hence, many companies are using this architectural style to develop their systems; for example, Netflix, Amazon, eBay, and Uber. The microservices architecture is an approach for developing an application as a set of small, well-defined, cohesive, loosely coupled, independently deployable, and distributed services. Microservices interact via messages, using standard data formats and protocols and published interfaces using a well-defined lightweight mechanism such as REST [9]. An important aspect of this architecture is that each microservice owns its domain model (data, logic, and behavior). Related functionalities are combined into a single business capability (called bounded context), and each microservice implements one such capability (one or several services) [24]. The microservices architecture assists in tackling the complexity of large applications by decomposing them into small pieces, where each component resides within its own bounded context.

This architecture also enables traceability between the requirements and the system structure, and thus only one microservice has to be changed and redeployed in order to update a domain [10].

The development of the microservices architecture aims to overcome shortcomings of *monolithic* architecture styles, where the user interface, the business logic, and the databases are packaged into a single application and deployed to a server. Whereas deployment is easy, a large monolithic application can be difficult to understand and to maintain, because once the system evolves, its modularity is eroded. Besides, every change causes the redeployment of the whole system.

Two important issues which are favored by the microservices community as keys to building a successful microservices architecture are the functional decomposition of an application and the decentralized governance (i.e., each service usually manages its unique database). One of the big challenges in designing the microservices architecture is to find an appropriate partition of the system into microservices [28]. For example, the microservice architecture can significantly affects the performance of the system [20]. It seems reasonable that each service will have only a very limited set of responsibilities, preferable only one—the single responsibility principle [27]. Determining the microservice granularity influences the quality of service of the microservice application [16] and also the number of microservices. Nevertheless, there is a lack of systematic approaches that decide and suggest the microservice boundaries as described in more detail in the following section. Hence, microservices design is usually performed intuitively, based on the experience of the system designers. However, providing an inappropriate partition into services and getting service boundaries wrong can be costly [29].

In this paper we describe a systematic approach to identify microservices in early design phase. We identify the relationships between the required system operations and the state variables that those operations read or write. We then visualize the relationships between the system operations and state variables, thus we can recognize clusters of dense relationships. This provides a partition of the system's state space into microservices, such that the operations of each microservice access only the variables of that microservice. This decomposition guarantees strong cohesion of each microservice and low coupling between services.

The remainder of the paper is organized as follows. The state of the art is discussed in Sect. 2. We use the CoCoME [32] case study to motivate and demonstrate our approach; in Sect. 3 we present the CoCoME trading system. Our approach for identifying microservices is described in Sect. 4. Systems evolve over time, and in Sect. 5 we describe the KAMP approach for change impact analysis which we will use for system maintenance. In Sect. 6, we evaluate our approach. We conclude in Sect. 7.

2 State of the Art

We now present the state of the art on identifying microservices. In the approach proposed by Newman [29], bounded contexts (i.e., responsibility have explicit boundaries) play a vital role in defining loosely coupled and high cohesive services. However, the question about how to systematically find those contexts remains open.

Use-cases are mentioned in [28] as an obvious approach to identify the services. Others, e.g. [14], suggest a partition strategy based on verbs. Some approaches for partitioning a system into microservices are described in [34]. Those approaches include: using nouns or resources, by defining a service that is responsible for all operations on entities or resources of a given type; decomposing by verbs or use cases and define services that are responsible for particular actions; decomposing by business capability, where a business capability is something that a business does in order to generate value; and decomposing by domain-driven design subdomain, where the business consists of multiple subdomains, each corresponding to a different part of the business. The domain-driven design approach [10] seems to be the most common technique for modeling microservices.

Some of the approaches listed in [34] are relevant to our approach (e.g., using the use cases, nouns, verbs). However, no systematic approach is offered. Baresi et al. [2] propose an automated process for finding an adequate granularity and cohesiveness of microservices candidates. This approach is based on the semantic similarity of foreseen/available functionality described through OpenAPI specifications (OpenAPI is a language-agnostic, machine-readable interface for REST APIs). By leveraging a reference vocabulary, this approach identifies potential candidate microservices, as fine-grained groups of cohesive operations (and associated resources). A systematic architectural modeling and analysis for managing the granularity of the microservices and deciding on the boundaries of the microservices is provided in [16]. The authors claim that reasoning about microservice granularity at the architectural level can facilitate analysis of systems that exhibit heterogeneity and decentralized governance.

However, there hardly are any guidelines on what is a 'good' size of a microservice [13, 29]. Basically the suggestion is to refine too large services, without providing a metric that defines what too large means. Practical experience shows that the size of the microservices heavily differ from one system to another. There is also no rigorous analysis of the actual dependencies between the system's functionality and its structure.

3 Running Example: CoCoME

To demonstrate our approach we have applied the CoCoME (Common Component Modeling Example) case study on software architecture modeling [19, 32]. It represents a trading system as can be observed in a supermarket chain handling sales, and is widely used as a common case study for software architecture modeling and evolution [18]. The example includes processing sales at a single store of the chain, e.g. scanning products or paying, as well as enterprise-wide administrative tasks, e.g. inventory management and reporting.

The system specification includes functional requirements for: selling products, ordering products from providers, receiving the ordered products, showing reports, and managing the stocks in each store. The specification is informal, and is given in terms of detailed use cases (in the format proposed by Cockburn [5]). In the following, we provide an excerpt of the use cases of CoCoME, as depicted in Fig. 1. A fully detailed description can be found in [32].

- *Process Sale*: this use case detects the products that a customer has purchased and handles the payment (by credit or cash) at the cash desk.
- *Order Products*: this use case is employed to order products from suppliers.
- *Receive Ordered Products*: this use case describes the requirement that once the products arrive at the store, their correctness have to be checked and the inventory has to be updated.
- *Show Stock Reports*: this use case refers to the requirement of generating stock-related reports.
- *Show Delivery Reports*: calculate the mean times a delivery takes.
- *Change Price*: describes the case where the sale price of a product is changed.

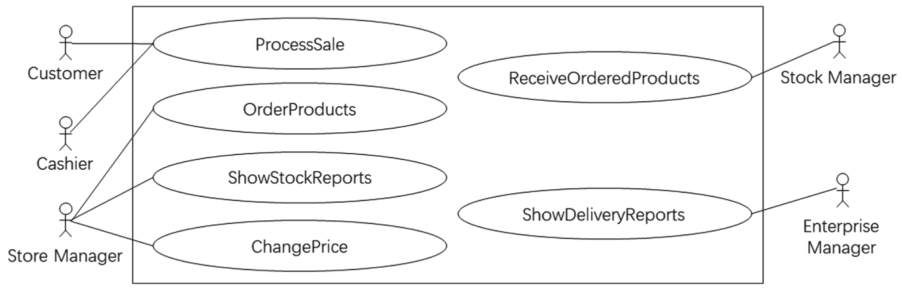


Fig. 1. The UML use case diagram for the CoCoME system.

4 Identifying Microservices

The proposed analytical approach to identify microservices described in this section is based on use case specification of the software requirements and on a functional decomposition of those requirements. To employ the suggested approach, we first need to create a model of the system. This model consists of a finite set of *system operations* and of the system's *state space*. System operations are the public operations (methods) of the system; i.e., the operations that comprise the system's API and which provide the system response to external triggers. The state space is the set of system variables which contain the information that system operations write and read.

System Decomposition. The decomposition of the system into microservices is achieved by partitioning the state space in a way that guarantees that each microservice has its own state space and operations. That is, the microservices partition the system state variables into disjoint sets such that the operations of each microservice may directly access only its local variables. When a microservice needs to refer (read or write) to a variable in another state space, it is achieved only via the API of the relevant microservice, i.e., the one that includes the relevant state space. This enables the selection of a good decomposition—i.e., one that guarantees low coupling as well as high cohesion. Thus, we model a system decomposition into microservices as a syntactical partition of the state space. A system is then built as a composition of microservices by conjoining their operations.

System requirements can be given in many forms, ranging from informal natural language to fully formal models. Even an existing implementation of the system (e.g., a monolith one) can serve as a source of requirements (see, e.g., [7], [12]). Use case specifications are widely accepted as a way of specifying functional requirements [21]. A use case describes how users (actors) employ a system to achieve a particular goal. We identify the *system operations* and the *system state variables* based on the description of the use cases. We record—in what we call *operation/relation table*—the relationships between each system operation and the state variables that the operation used (reads or writes). That is, each cell in the table indicates whether the operation writes to the state variable, reads it, or neither writes to it nor reads it. In order to identify the system operations and system state variables, we find—as a first approximation[1]—all the verbs in the informal descriptions of the use cases. The nouns found in those descriptions serve as an approximation for the system state variables [1][2]. Note that our approach works in general once the operations and state variables are identified, without the need to know how they are gathered. Nevertheless, we shortly describe how we have collected the information that is used to create the operation/relation table, as it makes the process even more systematic, compared to any ad-hoc approach of extracting variables and operations. We use tools (e.g., easyCRC [31], TextAnalysisOnline [39]) that assist us to extract nouns and noun phrases from the use case specifications (as candidates for state variables) as well as verbs (suggesting system operations). This process, however, can be done without using any tool. This systematic approach enables us to identify operations based on the use case descriptions and to produce informal and formal specification of the contracts of the system operations. This then allows us to carry out formal analysis and validation of the system model [25], as discussed in Sect. 6.

Visualization. The operation/relation table is then visualized, shown in a graph form. The visualization as a graph enables us to identify clusters of dense

[1] The list of verbs that we have found may be updated as some verbs may not be system operations, others may not be mentioned in the informal description, and some verbs are synonyms, hence they describe the same operation.

[2] A brain storming is needed to handle issues such as synonyms, irrelevant nouns, etc.

relationships that are weakly connected to other clusters.[3] Each such cluster is considered a good candidate to become a microservice, because:

1. The amount of information it shares with the rest of the system is small, thus it is protected from changes in the rest of the system and vice versa—this satisfies the low coupling requirement.
2. The internal relationships are relatively dense, which in most cases indicates a cohesive unit of functionality, satisfying the demand for strong cohesion.

We build an undirected bipartite graph G whose vertices represent the system's state variables and operations. An edge connects an operation *op* to a state variable v if and only if *op* either reads the value of v or updates it. In addition, we assign a weight to each edge of G, depending on the nature of the connection. A read connection has a lower weight (we have chosen 1) and a write connection has a higher weight (in our case 2). This choice tends to cluster together data with those operations that change it, preferring read interfaces between clusters. A write interface actively engages both subsystems, thus it has a stronger coupling. While different numbers are possible for the weights, the chosen numbers result in a graph that satisfies our needs to identify clearly separated clusters. Note, however, that we have also tried other weights—yet keeping the weight of the write operation higher than that of the read operation. Whereas this sometimes has changed the layout of the graph, it did not change the clustering. For the visualization of the graph we use `NEATO` [30]—a spring model based drawing algorithm. The program draws undirected graphs such that nodes that are close to each other in graph-theoretic space (i.e. shortest path) are drawn closer to each other. The left hand side of Fig. 2 presents an example of the operation/relation dependency graph of the CoCoME system, as drawn by `NEATO`. Note that this visualization can be used in various ways: to suggest low dependency partitions, where each part can serve as a microservice; to evaluate partitions into microservices that are dictated by non-functional constraints; and to explore changes to the system model that reduce the dependencies between areas that we consider as good microservices candidates. We have used the partition into clusters to identify the possible microservices. The right hand side of Fig. 2 describes the microservices that we have identified.

At this point it is important to emphasize that the idea of software clustering is not a new one; the reader may refer, for example, to [8, 26]. Those works investigate clustering based on source code analysis, and the main idea is to enable developers to understand the structure of evolving software. The source level modules and dependencies are mapped to a module dependency graph and then the graph is partitioned so that closely related nodes are grouped into composite nodes—the clusters (subsystems). However, who guarantees that the design of the developed system was 'good' with respect to strong cohesiveness and weak coupling? Quoting [26]: "Creating a good mental model of the structure of a complex system, and keeping that mental model consistent with changes

[3] A clustering of a graph G consists of a partition of the node set of G.

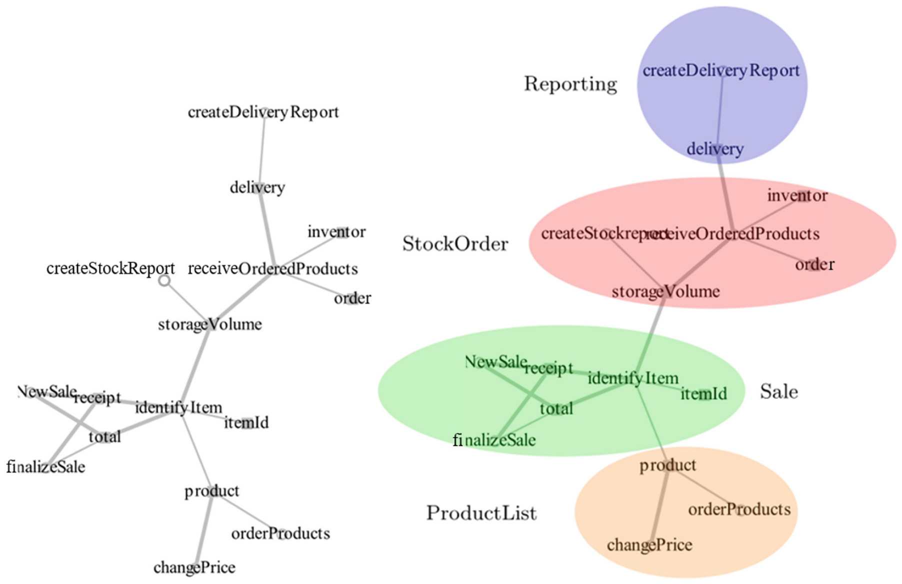


Fig. 2. An operation/relation dependency graph of CoCoME. The left side shows the diagram before identifying the microservices, and the right side presents also the microservices (the colored shapes). Thin/thick edges represent read/write relationship; circles represent operations; and squares represent state variables. Note that the graph was created by NEATO; the truncation in names (e.g. order) was done by NEATO. (Color figure online)

that occur as the system evolves, is one of many serious problems that confront today's software developers. ... we have developed an automatic technique to decompose the structure of software systems into meaningful subsystems. Subsystems provide developers with high-level structural information that helps them navigate through the numerous software components, their interfaces, and their interconnections".

Finding the clusters at the source code level definitely helps to understand the structure of the system. But it may be the case that the design is bad with regard to coupling and cohesion. Correcting this once the code exists is sometimes a very difficult task. Using clustering before the code exists, as done in our approach, enables to develop software that is of higher design quality. It is also easier than to maintain the software, for example with tools like KAMP, as elaborated in Sect. 5. Moreover, having both the operation/relation table and the clustering that has been obtained using this table, traceability is much easier. Suppose, for example, that the user has to change a function; then she can easily recognize in which component this function is implemented.

APIs and Databases. For each identified microservice we define its API and its data storage. The API of the microservice (cluster) is provided by the union of the system operations which write into the state variables belonging to the cluster that has been identified based on the visualization process. When a system operation of another cluster reads information from the current cluster, a getter method is added to the API of the current cluster; i.e., the access is only through a published service interface. For example, the operation *identifyItem* is part of the API of the *Sale* microservice (see the right hand side of Fig. 2). Since *identifyItem* needs to read the *product* state variable of the *ProductList* microservice, the *getProduct* operation is added to *ProductList*'s API.

There are two basic approaches regarding using databases for microservices. (i) Hasselbring and Steinacker [17] propose the *share nothing* approach according to which each microservice should have its own database. The advantages of this approach is higher speed, horizontal scalability, and improved fault-tolerance. One can also use a polyglot persistence architecture, where for each service the best suited type of database is chosen. However, this approach is at the price of data consistency, since consistency may only be eventual consistency (see [37]), and problems are dealt with by compensating operations. (ii) Yanaga [38] as well as Lewis and Fowler [24] claim that information can be shared. Yanaga argues that since a microservice is not an island, the data between services has to be integrated. That is, a microservice may require information provided by other services and provides information required by other services.

We agree that services sometimes need to share data. Nevertheless, this sharing should be minimal, to make the microservices as loosely coupled as possible. In our approach we analyze each created cluster. The information that needs to be persistent is found in the various clusters. If needed, we add to the cluster (i.e., microservice) a database that contains the persistent data that is private to this specific service. Of course it might be that the persistent data is located in different clusters. In this case we may end up with several microservices that contain data that is needed by other services. We guarantee that those databases are accessible only via the API of the services that contain them; i.e., no direct database access is allowed from outside the service.

Approach Summary. Our approach can be summarized as follows:

1. Analyze the use case specifications (write out their detailed descriptions if needed).
2. Identify the system operations and the system state variables (based on the use cases and their scenarios).
3. Create an *operation/relation* table.
4. Advise a possible decomposition into high cohesive and low coupled components (using a visualization tool).
5. Identify the microservices APIs.
6. Identify the microservices databases.
7. Implement the microservices (using RESTish protocols as the means of communication between microservices).
8. Deploy.

We have not referred to implementation and deployment; this is done in Sect. 6.

5 Architecture-Based Change Impact Analysis

Software systems must evolve over time, since otherwise they progressively become less useful [23]. Once a system is released, it continuously changes, e.g. due to emerging user requirements (perfective changes), bug fixes (corrective changes), platform alterations (adaptive changes), or correction of latent faults before they become operational faults (preventive changes). Consequently, the system drifts away from its initial architecture due to the evolutionary changes; yet, knowledge about the software architecture is essential to predict maintenance efforts.

The KAMP approach [35] supports software architects assessing the effect of change requests on technical and organizational work areas during software evolution. Based on the Palladio Component Model (for details see [33]), KAMP supports modeling the initial software architecture—the *base architecture*, and the architecture after a certain change request has been implemented in the model—the *target architecture*. The KAMP tooling calculates the differences between the base and the target architecture models and analyses the propagation of changes in the software system. Due to this change propagation analysis, services affected by a given change request can be identified. A large number of affected services may indicate the necessity for redesigning the system. In such a case we have to update the operation table according to the new requirements, and to continue in the process as described in this paper.

6 Evaluation

In this section, we exemplify our approach based on the CoCoME community case study [32] and evaluate our results.

Case Study: Starting with the use case specification of CoCoME, we identify the system operations as well as their state variables. Table 1 shows the operation/relation table that has been created based on this information. This table is then visualized as a graph, shown on the left hand side of Fig. 2. Based on the graphical representation we have identified four major clusters which are candidates to become microservices: *ProductList*, *Sale*, *StockOrder*, and *Reporting*; see the right hand side of Fig. 2. In this way we have achieved a meaningful partition into clusters, where each cluster has high cohesion and the coupling between the clusters is low. Each of the four identified microservice candidates (clusters) is responsible for a single bounded context in a business domain [10]. As can be seen on the right hand side of Fig. 2, the microservices are not totally independent. The communication between the microservices is implemented using REST APIs [29].

Note that the emphasize in our approach is on identifying microservices that deal with one business capability, rather than minimizing the microservices size; this conforms to [6]. Moreover, as stated in [6], a clear cohesion based on a high number of dependencies indicates the need for a single microservice. This

Table 1. Operation/relation table for CoCoME

Operation	itemId	mode	receipt	product	total	storageVolume	order	creditDetails	inventory	delivery	suppliers
startNewSale			2		2						
identifyItem	1		2	1	2	2					
finalizeSale			2		1						
cardPayment								2			
changeMode		2									
orderProducts				1						2	1
receiveOrderedProducts						2	1		2	2	
createStockReport						1					
createDeliveryReport				1						1	
changePrice				2							

is exactly what our functional decomposition achieves, and it actually provides bounded context—as suggested in the domain-driven design approach [10].

Evaluation Goal and Design: To evaluate our approach, we have compared our proposed decomposition of the CoCoME system into microservices with the microservices that have been identified by three independent software projects that have independently implemented CoCoME; the implementers have not been aware of the other implementations. They also have not known our approach of system decomposition. One project has been developed in RISE Centre at Southwest University (SWU-RISE) in Chongqing, China, and the other two have been built in Karlsruhe Institute of Technology (KIT), Germany. All projects have been developed by students. The supervisors of the projects have not been involved in the microservices' design nor did they provide any hint that might have influenced the design. The goal of the evaluation is to check whether our approach provides a decomposition of the system into microservices that is similar to a decomposition suggested by humans. Of course it might be the case that the latter decomposition is wrong; therefore we compared the results to three implementations. Note that there may exist several decompositions of a given system, where each has its advantages and disadvantages. Thus, we cannot claim that the decomposition provided by our approach is *the* best one, while other decompositions are not as good. If our approach provides results that are comparable to the decompositions done by human developers, then it has the advantage that due to its systematic and tool-supported nature it provides the decomposition much faster than when done manually.

Two of the CoCoMe implementations have been developed in KIT. Two master students, working independently of each other, have decomposed the system into microservices as part of the requirements in a practical course on software engineering. The students have basic knowledge in software architecture styles and in developing microservices. The students decomposed the system into microservices merely based on the existing use case specification and on an existing source code of a service-oriented implementation of CoCoME. To identify the microservices, the students also had to understand the design documentation—a component diagram and sequence diagrams [19]. They have identified microservices candidates based on the domain-driven design [10]. Then they modeled the application domain and divided it into different bounded contexts, where each bounded context was a candidate to be a microservice. Later they have made explicit the interrelationships between the bounded contexts. After understanding the requirements, it was a matter of days for them to design the microservices architecture of the system. The design and implementations created by the students can be found on GitHub.[4]

Another group of students, in the RISE Centre at Southwest University (SWU-RISE), has also been involved in the evaluation of our approach. This team has been composed of three computer science students: two first year post graduate students and one undergraduate (senior student). The students have basic knowledge in software architecture styles and in developing microservices. All students have a basic idea of SOA, web services, and microservices-based systems. One postgraduate student also has more than one year experience in web-based system development and Docker usage. The team had a supervisor that worked with them to control the progress of the development and to provide consulting work on requirements (thus acting as the software client). The students have been responsible for all development phases: requirements analyzing and modeling, system design, and system implementation and deployment. The supervisor, however, was not involved in the actual design of the microservice, and as mentioned provided no hints that may have led to the design proposed in this paper. The following principles guided this team in their division into microservices:

- *Identify business domain objects.* For example, the CoCoME domain objects includes *Order*, *Commodity*, *Transaction*, *Payment*, etc. The aim of this principle is to find those microservices that correspond to one domain object. This process was done by analyzing the use case descriptions and building a conceptual class model [3].
- *Identify special business.* This is the case that a business process spans multiple domain objects, and then an independent microservice is designed.
- *Reuse.* Business processes that are frequently called may be detached from the object and designed as a single microservice. Accordingly, if a constructed microservice is seldom used or difficult to be implemented separately, it can be attached to some object microservice or other microservices.

The team used the rCOS approach [4] to analyze the requirements and design the microservices. The process can be summarized as follows: identify the use

[4] For the implementations see https://github.com/cocome-community-case-study.

cases (this step was not needed, as brief descriptions of the use cases specifications have been provided); construct the conceptual class diagram and use case sequence diagrams; for each use case operation write contracts in terms of its pre- and post-conditions. Those artifacts have formed a formal requirements model; analysis can then be applied to verify consistency and correctness [4]. The CoCoMe requirements have been analyzed and validated for consistency and functional correctness using the AutoPA tool [25]. It was a matter of days for the students to design the microservices architecture of the system.[5]

Evaluation Results: Although the students at KIT named the microservices differently from the names used by us, both implementations also created four microservices. Each microservice has the same functionality as the one that was provided by employing our approach:

- *ProductList*: Managing the products that are stored in the trading system.
- *Sale*: Handling a sale in the trading system.
- *StockOrder*: Managing a stock order of products.
- *Reporting*: Creating a report of the delivered products.

The microservices identified by the students at KIT are all connected to a Frontend-Service, which provides the basic panel in which the user interfaces of the microservices are displayed. Thus, the Frontend-Service serves as a single integration point, similar to controllers that are used in related approaches. Controllers are responsible for receiving and handling system operations [22]. The use case controller pattern, for example, deals with all system events of a use case. It suggests delegating the responsibility for managing a use case flow of execution to a specialized controller object, thus localizing this functionality and its possible changes. The controllers provide a uniform way of coordinating actor events, user interfaces, and system services.

The microservices that have been identified by the students at KIT and their relationship are depicted in Fig. 3.

The students at SWU-RISE identified eight microservices:

- *Inventory*: Handles the products.
- *Commodity*: Queries the information regarding the product to be sold.
- *Order*: Manages the orders.
- *Supplier*: Handles the suppliers.
- *Transaction*: Refers to the sale (creation, management, query and end of the transactions).
- *Supplier_evaluation*: Calculates average time for the supplier to deliver the product to the supermarket.
- *Inventory_report*: Produces reports.
- *Pay*: Recording payment records—cash or non-cash.

[5] The implementation created by this group of students can be found in http://cocome.swu-rise.net.cn/.

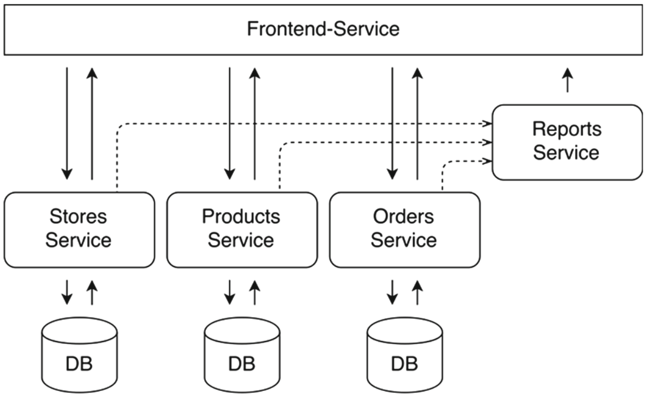


Fig. 3. Overview of the CoCoME microservices designed by KIT students [36]

The microservices that have been identified by the students at SWU-RISE and their relationship are depicted in Fig. 4. As can be seen, there are four different GUIs for the actors of CoCoME. The GUI combines a few microservices together to accomplish a specific function through a gateway.

The evaluation results show that the two teams in KIT have implemented the CoCoME model using the same decomposition as advised by our approach; the only difference is in the names of the microservices. The SWU-RISE team created eight microservices. However, a thorough examination reveals that the extra microservices are a refinement of the microservices provided by our approach. That is, each of the additional implemented microservices also appears as an activity in the coarser suggested decomposition. For example, the *Stock-Order* microservice refers to the variables *order* and *inventory* which became fine-grained microservices in the implementation of the students in SWU-RISE. Also the *Reporting* microservice has been split—into the *Supplier* and *Supplier_evaluation* microservices. This means that our approach can serve as the base decomposition. As mentioned in Sect. 4, the user can explore the visualization and decide whether to further break the system into finer microservices. Doing this results in exactly the decomposition provided by the SWU-RISE group, just using different microservices names. Moreover, recall that our goal is to provide a systematic and meaningful, high cohesive decomposition into microservices rather than finding the finest granular decomposition. A high cohesive cluster that is identified using our approach indicates the need for a microservice, and this microservice sometimes can be refined further. As recognized also in e.g. [6,15], a design problem of developing a system is to find the optimal level of granularity of a microservice architecture, and it is not always an immediate process. Balancing the size and the number of microservices is a design trade-off.

Our approach guarantees that each service can get what it needs from the other services. In contrast, when developing microservices without such a

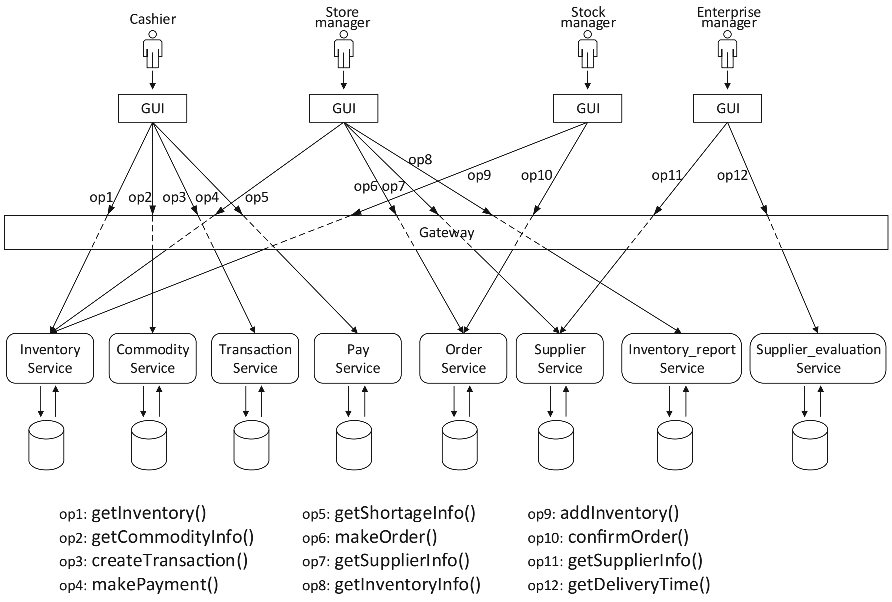


Fig. 4. Overview of the CoCoME microservices designed by SWU-RISE students

systematic approach, it is often difficult to understand and follow the interconnections between the services so thoroughly [29]. The evaluation results give us reasons to believe that our approach identifies microservice in a quality that is comparable to a design done by human software designers. Our approach, however, achieved the microservices identification much faster and with less effort compared to human developers. While identifying the microservices was a matter of days for the students at KIT and SWU-RISE, by employing our approach it was a matter of hours. Moreover, the problem of identifying the appropriate microservices becomes much more complicated as the system becomes more complex. A large real world system has much more details, so the chances of getting an appropriate decomposition by intuition will decrease.

7 Conclusion

We proposed a systematic and practical engineering approach to identify microservices which is based on use-case specification and functional decomposition of those requirements. This approach provides high cohesive and low coupled decomposition. To evaluate our approach, we have compared the results to three independent implementations of the CoCoME system. The evaluation give us reasons to believe in the potential of our approach. We will involve other kind of systems in the evaluation. Doing this we will also examine the scalability of our approach. We believe it is scalable, since the tools that create the diagrams and suggest decompositions are quite fast and can work on large graphs [11].

Moreover, if indeed as in the CoCoME model each use case is handled in only one component provided in the decomposition, then the work can be split among distributed teams.

Acknowledgment. This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593, and the Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics) under the Open Foundation with No. NJ20170007. We would like to thank Kang Cheng, Guohang Guo, Yukun Zhang, Nils Sommer, and Stephan Engelmann who worked on the development of the various CoCoME systems. We also thank the anonymous reviewers for their careful reading and their many insightful comments and suggestions. Their comments helped to improve and clarify this paper.

References

1. Abbott, R.J.: Program design by informal english descriptions. Commun. ACM **26**(11), 882–894 (1983)
2. Baresi, L., Garriga, M., De Renzis, A.: Microservices identification through interface analysis. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 19–33. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_2
3. Chen, X., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75698-9_13
4. Chen, Z., et al.: Refinement and verification in component-based model-driven design. Sci. Comput. Program. **74**(4), 168–196 (2009)
5. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, Boston (2000)
6. de la Torre, C., et al.: .NET Microservices: Architecture for Containerized .NET Applications. Microsoft (2017)
7. De Santis, S., et al.: Evolve the Monolith to Microservices with Java and Node. IBM Redbooks, Armonk (2016)
8. Doval, D., Mancoridis, S., Mitchell, B.S.: Automatic clustering of software systems using a genetic algorithm. In: STEP, pp. 73–81. IEEE (1999)
9. Dragoni, N., et al.: Microservices: yesterday, today, and tomorrow. Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_12
10. Evans, E.: Domain Driven Design: Tackling Complexity in the Heart of Business Software. Addison-Wesley, Boston (2004)
11. Faitelson, D., Tyszberowicz, S.: Improving design decomposition. Form. Asp. Comput. **22**(1), 5:1–5:38 (2017)
12. Fowler, M.: MonolithFirst (2015). https://martinfowler.com/bliki/MonolithFirst.html#footnote-typical-monolith. Accessed Mar 2018
13. Francesco, P.D., et al.: Research on architecting microservices: trends, focus, and potential for industrial adoption. In: ICSA, pp. 21–30. IEEE (2017)
14. Hassan, M., Zhao, W., Yang, J.: Provisioning web services from resource constrained mobile devices. In: IEEE CLOUD, pp. 490–497 (2010)
15. Hassan, S., Bahsoon, R.: Microservices and their design trade-offs: a self-adaptive roadmap. In: SCC, pp. 813–818. IEEE (2016)

16. Hassan, S., et al.: Microservice ambients: an architectural meta-modelling approach for microservice granularity. In: ICSA, pp. 1–10. IEEE (2017)
17. Hasselbring, W., Steinacker, G.: Microservice architectures for scalability, agility and reliability in E-Commerce. In: ICSA Workshops, pp. 243–246. IEEE (2017)
18. Heinrich, R., et al.: A platform for empirical research on information system evolution. In: SEKE, pp. 415–420 (2015)
19. Heinrich, R., et al.: The CoCoME platform for collaborative empirical research on information system evolution. Technical report 2016:2, KIT, Germany (2016)
20. Heinrich, R., et al.: Performance engineering for microservices: research challenges and directions. In: Companion Proceedings of ICPE, pp. 223–226 (2017)
21. Jacobson, I., et al.: Object-Oriented Software Engineering - A Use Case Driven Approach. Addison-Wesley, Boston (1992)
22. Larman, C.: Applying UML and Patterns, 3rd edn. Prentice Hall, Upper Saddle River (2004)
23. Lehman, M.M.: On understanding laws, evolution, and conservation in the large-program life cycle. J. Syst. Softw. **1**, 213–221 (1980)
24. Lewis, J., Fowler, M.: Microservices. https://martinfowler.com/articles/microservices.html. Accessed Apr 2018
25. Li, X., Liu, Z., Schäf, M., Yin, L.: AutoPA: automatic prototyping from requirements. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010. LNCS, vol. 6415, pp. 609–624. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16558-0_49
26. Mancoridis, S., et al.: Bunch: a clustering tool for the recovery and maintenance of software system structures. In: ICSM, pp. 50–59. IEEE Computer Society (1999)
27. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, Upper Saddle River (2003)
28. Namiot, D., Sneps-Sneppe, M.: On micro-services architecture. J. Open Inf. Technol. **2**(9), 24–27 (2014)
29. Newman, S.: Building Microservices. O'Reilly, Sebastopol (2015)
30. North, S.C.: Drawing graphs with NEATO. User Manual (2004)
31. Raman, A., Tyszberowicz, S.S.: The EasyCRC tool. In: ICSEA, pp. 52–57. IEEE (2007)
32. Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.): The Common Component Modeling Example: Comparing Software Component Models. LNCS, vol. 5153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85289-6
33. Reussner, R.H., et al.: Modeling and Simulating Software Architectures - The Palladio Approach. MIT Press, Cambridge (2016)
34. Richardson, C.: Microservices from design to deployment (2016). https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/
35. Rostami, K., Stammel, J., Heinrich, R., Reussner, R.: Architecture-based assessment and planning of change requests. In: QoSA, pp. 21–30 (2015)
36. Sommer, N.: Erweiterung und Wartung einer Cloud-basierten JEE-Architektur (in German), report of a practical course. Technical report, KIT, Germany (2017)
37. Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)
38. Yanaga, E.: Migrating to Microservice Databases: From Relational Monolith to Distributed Data. O'Reilly, Sebastopol (2017). E-book
39. Text analysis. http://textanalysisonline.com/textblob-noun-phrase-extraction. Accessed Apr 2018

Verification

Robust Non-termination Analysis of Numerical Software

Bai Xue[1](✉), Naijun Zhan[1,2](✉), Yangjia Li[1,3](✉), and Qiuye Wang[1,2](✉)

[1] State Key Laboratory of Computer Science, Institute of Software, CAS, Beijing, China
{xuebai,znj,yangjia,wangqy}@ios.ac.cn

[2] University of Chinese Academy of Sciences, CAS, Beijing, China

[3] University of Tartu, Tartu, Estonia

Abstract. Numerical software is widely used in safety-critical systems such as aircrafts, satellites, car engines and many other fields, facilitating dynamics control of such systems in real time. It is therefore absolutely necessary to verify their correctness. Most of these verifications are conducted under ideal mathematical models, but their real executions may not follow the models exactly. Factors that are abstracted away in models such as rounding errors can change behaviors of systems essentially. As a result, guarantees of verified properties despite the present of disturbances are needed. In this paper, we attempt to address this issue of nontermination analysis of numerical software. Nontermination is often an unexpected behaviour of computer programs and may be problematic for applications such as real-time systems with hard deadlines. We propose a method for robust conditional nontermination analysis that can be used to under-approximate the maximal robust nontermination input set for a given program. Here robust nontermination input set is a set from which the program never terminates regardless of the aforementioned disturbances. Finally, several examples are given to illustrate our approach.

Keywords: Numerical software · Nontermination analysis
Robust verification

1 Introduction

Software is ubiquitous in mission-critical and safety-critical industrial infrastructures as it is, in principle, the most effective way to manipulate complex systems in real time. However, many computer scientists and engineers have experienced

This work from Bai Xue is funded by CAS Pioneer Hundred Talents Program under grant No. Y8YC235015, and from Naijun Zhan is funded partly by NSFC under grant No. 61625206 and 61732001, by "973 Program" under grant No. 2014CB340701, and by the CAS/SAFEA International Partnership Program for Creative Research Teams, and from Yangjia Li is funded by NSFC under grant No. 61502467.

costly bugs in embedded software. Examples include the failure of the Ariane 5.01 maiden flight (due to an overflow caused by an unprotected data conversion from a too large 64-bit floating point to a 16-bit signed integer value), the failure of the Patriot missile during the Gulf war (due to an accumulated rounding error), the loss of Mars orbiter (due to a unit error). Those examples indicate that mission-critical and safety-critical software may be far from being safe [11]. It is therefore absolutely necessary to prove the correctness of software by using formal, mathematical techniques that enable the development of correct and reliable software systems.

The dominant approach to the verification of programs is called *Floyd-Hoare-Naur inductive assertion approach* [13,18,34]. It uses *pre-* and *post-condition* to specify the condition of initial states and the property that should be satisfied by terminated states, and use *Hoare logic* to reason about properties of programs. The hardest parts of this approach are *invariant generation* and *termination analysis*. It is well-know that the termination or nontermination problem is undecidable, and even not semi-decidable in general. Thus, more practical approaches include present some sufficient conditions for termination, or some sufficient conditions for nontermination, or put these two types of conditions in parallel, or prove the decidability for some specific families of programs, e.g., [3,7,14,16,23,25,39,53].

On the other hand, most of these verifications are conducted under ideal mathematical models, but their real executions may not follow the models exactly. Factors that are abstracted away in models such as rounding errors can change behaviors of systems essentially. As a result, guarantees of verified properties despite the present of disturbances are needed. We notice that this problem affects most existing termination/nontermination as well.

In [54], the authors presented the following example:

Example 1. Consider a simple loop

$$\text{Q1:}\quad \textbf{while } (B\mathbf{x} > \mathbf{0})\ \{\mathbf{x} := A\mathbf{x}\},$$

where $A = \begin{pmatrix} 2 & -3 \\ -1 & 2 \end{pmatrix}$, $B = \begin{pmatrix} 1 & b \\ -1 & b \end{pmatrix}$ with $b = -\frac{1127637245}{651041667} = -\sqrt{3} + \epsilon \sim -1.732050807$.

So we have $\epsilon = \sqrt{3} - (-\frac{1127637245}{651041667}) > 0$ a small positive number. Here we take 10 decimal digits of precision.

According to the termination decidability result on simple loops proved in [48], Q1 terminates based on exact computation. But unfortunately, it does not terminate in practice as the core decision procedure given in [48] invokes a procedure to compute *Jordan normal form* based on numeric computation for a given matrix, and thus the floating error has to be taken into account. In order to address this issue, a symbolic decision procedure was proposed in [54]. However, a more interesting and challenging issue is to find a systematic way to take all possible disturbances into account during conducting termination and non-termination analysis in practical numerical implementations.

In this paper we attempt to address this challenge, and propose a framework for robust nontermination analysis for numerical software based on control theory as in [40], which proposes a control-theoretic framework based on Lyapunov invariants to conduct verification of numerical software. Non-termination analysis proves that programs, or parts of a program, do not terminate. Non-termination is often an unexpected behaviour of computer programs and implies the presence of a bug. If a nonterminating computation occurs, it may be problematic for applications such as real-time systems with hard deadlines or situations when minimizing workload is important. In this paper, computer programs of interest are restricted to a class of computer programs composed of a single loop with a complicated switch-case type loop body. These programs can also be viewed as a constrained piecewise discrete-time dynamical system with time-varying uncertainties. We reformulate the problem of determining robust conditional nontermination as finding the maximal robust nontermination input set of the corresponding dynamical system, and characterize that set using a value function, which turns out to be a solution to a suitable mathematical equation. In addition, when the dynamics of the piecewise discrete-time system in each mode is polynomial and the state and uncertain input constraints are semi-algebraic, the optimal control problem is relaxed as a semi-definite programming problem, to which its polynomial solution forms an inner-approximation of the maximal robust nontermination input set when exists. Such relaxation is sound but incomplete. Finally, several examples are given to illustrate our approach.

It should be noticed that the concept of robust nontermination input sets is essentially equivalent to the maximal robustly positive invariants in control theory. Interested readers can refer to, e.g., [2,40,45,47]. Computing the maximal robustly positive invariant of a given dynamical system is still a long-standing and challenging problem not only in the community of control theory. Most existing works on this subject focus on linear systems, e.g. [21,38,45,47,51]. Although some methods have been proposed to synthesize positively invariants for non-linear systems, e.g., the barrier certificate generation method as in [36,37] and the region of attraction generation method as in [15,19,30,49]. These methods, however, resort to bilinear sum-of-squares programs, which are notoriously hard to solve. In order to solve the bilinear sum-of-squares programs, a commonly used method is to employ some form of alteration (e.g., [19,30,50]) with a feasible initial solution to the bilinear sum-of-squares program. Recently, [43,44] proposed linear programming based methods to synthesize maximal (robustly) positive polyhedral invariants. Contrasting with aforementioned methods, in this paper we propose a semi-definite programming based method to compute semi-algebraic invariant. Our method does not require an initial feasible solution.

Organization of the Paper. The structure of this paper is as follows. In Sect. 2, basic notions used throughout this paper and the problem of interest are introduced. Then we elucidate our approach for performing conditional non-termination analysis in Sect. 3. After demonstrating our approach on several illustrating examples in Sect. 4, we discuss related work in Sect. 5 and finally conclude this paper in Sect. 6.

2 Preliminaries

In this section we describe the programs which are considered in this paper and we explain how to analyze them through their representation as piecewise discrete-time dynamical systems.

The following basic notations will be used throughout the rest of this paper: $\mathbb{N}$ stands for the set of nonnegative integers and $\mathbb{R}$ for the set of real numbers; $\mathbb{R}[\cdot]$ denotes the ring of polynomials in variables given by the argument, $\mathbb{R}_d[\cdot]$ denotes the vector space of real multivariate polynomials of degree d, $d \in \mathbb{N}$. Vectors are denoted by boldface letters.

2.1 Computer Programs of Interest

In this paper the computer program of interest, as described in Program 1, is composed of a single loop with a possibly complicated switch-case type loop body, in which variables $\boldsymbol{x} = (x_1, \dots, x_n)$ are assigned using parallel assignments $(x_1, \dots, x_n) := \boldsymbol{f}(x_1, \dots, x_n, d_1, \dots, d_m)$, where $\boldsymbol{d} = (d_1, \dots, d_m)$ is the vector of uncertain inputs, of which values are sampled nondeterministically from a compact set, i.e. $(d_1, \dots, d_m) \in D$, such as round-off errors in performing computations. The form of programs under consideration is given in Program 1. In Program 1, $D = \{\boldsymbol{d} \mid \bigwedge_{i=1}^{n_{k+1}} h_{k+1,i}(\boldsymbol{d}) \leq 0\}$ is a compact set in $\mathbb{R}^m$ and $h_{k+1,i} : \mathbb{R}^m \mapsto \mathbb{R}$, is continuous over $\boldsymbol{d}$. $\Omega \subseteq \mathbb{R}^n$ stands for the initial condition on inputs; $X_0 = \{\boldsymbol{x} \in \mathbb{R}^n \mid \bigwedge_{i=1}^{n_0} [h_{0,i}(\boldsymbol{x}) \leq 0]\}$ stands for the loop condition, which is a compact set in $\mathbb{R}^n$; $X_j = \{\boldsymbol{x} \in \mathbb{R}^n \mid \bigwedge_{i=1}^{n_j} [h_{j,i}(\boldsymbol{x}) \rhd 0]\}$, $j = 1, \dots, k$, stands for the j-th branch conditions, where $\rhd \in \{\leq, <\}$. $h_{j,i} : \mathbb{R}^n \mapsto \mathbb{R}$, $j = 0, \dots, k$, $i = 1, \dots, n_j$, $\boldsymbol{f}_l : \mathbb{R}^n \times D \mapsto \mathbb{R}^n$, $l = 1, \dots, k$, are continuous functions over $\boldsymbol{x}$ and over $(\boldsymbol{x}, \boldsymbol{d})$ respectively. Moreover, $\{X_1, \dots, X_k\}$ forms a complete partition of $\mathbb{R}^n$, i.e. $X_i \cap X_j = \emptyset$ for $\forall i \neq j$, where $i, j \in \{1, \dots, k\}$, and $\cup_{j=1}^{k} X_j = \mathbb{R}^n$.

Program 1. Computer Programs of Interest

```
x := x_0;/* $\boldsymbol{x}_0 \in \Omega$                                   */
while $\boldsymbol{x} \in X_0$ do
    /* $\boldsymbol{d} \in D$                                              */
    if $\boldsymbol{x} \in X_1$ then
        $\boldsymbol{x} := \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{d})$;
    end
    else if $\boldsymbol{x} \in X_2$ then
        $\boldsymbol{x} := \boldsymbol{f}_2(\boldsymbol{x}, \boldsymbol{d})$;
    end
    ...
    else if $\boldsymbol{x} \in X_k$ then
        $\boldsymbol{x} := \boldsymbol{f}_k(\boldsymbol{x}, \boldsymbol{d})$;
    end
end
```

As described in Program 1, an update of the variable $\boldsymbol{x}$ is executed by the i-th branch $\boldsymbol{f}_i : \mathbb{R}^n \times D \mapsto \mathbb{R}^n$ if and only if the current value of $\boldsymbol{x}$ satisfies the i-th branch condition X_i.

2.2 Piecewise Discrete-Time Systems

In this subsection we interpret Program 1 as a constrained piecewise discrete-time dynamical system with uncertain inputs. Formally,

Definition 1. *A constrained piecewise discrete-time dynamical system (PS) is a quintuple* $(\boldsymbol{x}_0, X_0, \mathcal{X}, D, \mathcal{L})$ *with*

- $\boldsymbol{x}_0 \in \Omega$ *is the condition on initial states;*
- $X_0 \subseteq \mathbb{R}^n$ *is the domain constraint, which is a compact set. A path can evolve complying with the discrete dynamics only if its current state is in* X_0*;*
- $\mathcal{X} := \{X_i, i = 1, \ldots, k\}$ *with* X_i *as interpreted in* Program 1*;*
- $D \subseteq \mathbb{R}^m$ *is the set of uncertain inputs;*
- $\mathcal{L} := \{\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{d}), i = 1, \ldots, k\}$ *is the family of the continuous functions* $\boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{d}) : X_i \times D \mapsto \mathbb{R}^n$.

In order to enhance the understanding of PS, we use the following figure, i.e. Fig. 1, to illustrate it further. From now on, we associate a PS representation to each program of the form Program 1. Since a program may admit several PS representations, we choose one of them, but the choice does not change the results provided in this paper.

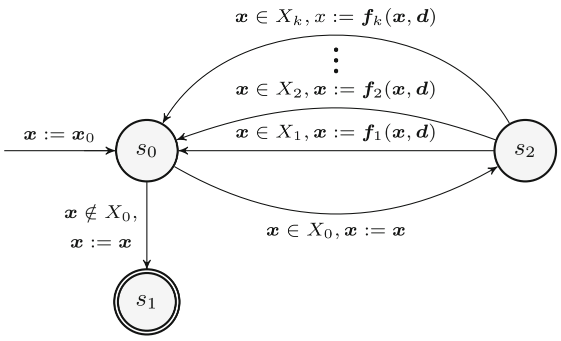

$\boldsymbol{x} \in X_k, x := \boldsymbol{f}_k(\boldsymbol{x}, \boldsymbol{d})$
$\boldsymbol{x} \in X_2, \boldsymbol{x} := \boldsymbol{f}_2(\boldsymbol{x}, \boldsymbol{d})$
$\boldsymbol{x} \in X_1, \boldsymbol{x} := \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{d})$
$\boldsymbol{x} := \boldsymbol{x}_0$
s_0
s_2
$\boldsymbol{x} \notin X_0,$
$\boldsymbol{x} := \boldsymbol{x}$
$\boldsymbol{x} \in X_0, \boldsymbol{x} := \boldsymbol{x}$
s_1

Fig. 1. An illustrating graph of PS

Definition 2. *An input policy* π *is an ordered sequence* $\{\boldsymbol{\pi}(i), i \in \mathbb{N}\}$*, where* $\boldsymbol{\pi}(\cdot) : \mathbb{N} \mapsto D$*, and* Π *is defined as the set of input policies, i.e.* $\Pi = \{\pi \mid \boldsymbol{\pi}(\cdot) : \mathbb{N} \mapsto D\}$.

If an input policy π makes Program 1 non-terminated from an initial state $\boldsymbol{x}_0$, then the trajectory $\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0} : \mathbb{N} \mapsto \mathbb{R}^n$ from $\boldsymbol{x}_0$ following the discrete dynamics is defined by

$$\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(l+1) = \boldsymbol{f}(\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(l), \boldsymbol{\pi}(l)), \tag{1}$$

where $\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(0) = \boldsymbol{x}_0$, $\forall l \in \mathbb{N}. \boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(l) \in X_0$, and

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{d}) = 1_{X_1} \cdot \boldsymbol{f}_1(\boldsymbol{x}, \boldsymbol{d}) + \cdots + 1_{X_k} \cdot \boldsymbol{f}_k(\boldsymbol{x}, \boldsymbol{d})$$

with $1_{X_i} : X_i \mapsto \{0, 1\}$, $i = 1, \ldots, k$, representing the indicator function of the set X_i, i.e.

$$1_{X_i} := \begin{cases} 1, & \text{if } \boldsymbol{x} \in X_i, \\ 0, & \text{if } \boldsymbol{x} \notin X_i. \end{cases}$$

Consequently, Program 1 is said to be robust nontermination starting from an initial state $\boldsymbol{x}_0 \in \Omega$ if for any input policy $\pi \in \Pi$, $\forall l \in \mathbb{N}.\ \boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(l) \in X_0$ holds. Formally,

Definition 3. *A program of* Program 1 *is said to be robust non-terminating w.r.t. an initial state $\boldsymbol{x}_0 \in X_0$, if*

$$\forall \pi \in \Pi.\ \forall l \in \mathbb{N}.\ \boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(l) \in X_0. \tag{2}$$

Now, we define our problem of deciding a set of initial states rendering Program 1 robust non-termination.

Definition 4 (Robust Nontermination Set). *A set Ω of initial states in $\mathbb{R}^n$ is a robust nontermination set for a program P of the form* Program 1 *if P is robustly non-terminating w.r.t. $\boldsymbol{x}_0$ for any $\boldsymbol{x}_0 \in \Omega$. We call $\{\boldsymbol{x}_0 \in \mathbb{R}^n \mid P$ is robustly non-terminating w.r.t. $\boldsymbol{x}_0\}$* the maximal robust non-termination set, *denoted by $\mathcal{R}_0$.*

From Definition 4, we observe that $\mathcal{R}_0$ is a subset of X_0 such that all runs of Program 1 starting from it can not breach it forever, i.e. if $\boldsymbol{x}_0 \in \mathcal{R}_0$, $\boldsymbol{f}(\boldsymbol{x}_0, \boldsymbol{d}) \in \mathcal{R}_0$ for $\forall \boldsymbol{d} \in D$. Therefore, the set $\mathcal{R}_0$ is equivalent to the maximal robust positively invariant for PS (1) in control theory. For the formal concept of maximal robust positively invariant, please refer to, e.g., [2,45,47].

3 Robust Non-termination Set Generation

In this section we elucidate our approach of addressing the problem of robust conditional nontermination for Program 1, i.e. synthesizing robust non-termination sets as presented in Definition 4. For this sake, we firstly in Subsect. 3.1 characterize the maximal robust non-termination set $\mathcal{R}_0$ by means of the value function, which is a solution to a mathematical equation. Any solution to this optimal control problem generates a robust non-termination set. Then, in the case that $\boldsymbol{f}_i$, $i = 1, \ldots, k$, is polynomial over $\boldsymbol{x}$ and $\boldsymbol{d}$, and the constraint sets over $\boldsymbol{x}$ and $\boldsymbol{d}$, i.e. X_j, $j = 0, \ldots, k$, and D, are of the basic semi-algebraic form, the semi-definite program arising from sum-of-squares decompositions facilitates the gain of inner-approximations Ω of $\mathcal{R}_0$ via solving the relaxation of the derived optimal control problem in Subsect. 3.2.

3.1 Characterization of $\mathcal{R}_0$

In this subsection, we firstly introduce the value function to characterize the maximal robust nontermination set $\mathcal{R}_0$ and then formulate it as a solution to a constrained optimal control problem.

For $\boldsymbol{x}_0 \in \mathbb{R}^n$, the value function $V : \mathbb{R}^n \mapsto \mathbb{R}$ is defined by:

$$V(\boldsymbol{x}_0) := \sup_{\pi \in \Pi} \sup_{l \in \mathbb{N}} \max_{j \in \{1,\ldots,n_0\}} \big\{h_{0,j}(\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(l))\big\}. \tag{3}$$

Note that $V(\boldsymbol{x}_0)$ may be neither continuous nor semi-continuous. (A function $V' : X' \mapsto \mathbb{R}$ is lower semicontinuous iff for any $y \in \mathbb{R}$, $\{\boldsymbol{x} \in X' \mid V'(\boldsymbol{x}) \geq y\}$ is open, e.g., [4].)

The following theorem shows the relation between the value function V and the maximal robust nontermination set $\mathcal{R}_0$, that is, the zero sublevel set of $V(\boldsymbol{x}_0)$ is equal to the maximal robust nontermination set $\mathcal{R}_0$.

Theorem 1. *$\mathcal{R}_0 = \{\boldsymbol{x}_0 \in \mathbb{R}^n \mid V(\boldsymbol{x}_0) \leq 0\}$, where $\mathcal{R}_0$ is the maximal robust nontermination set as in Definition 4.*

Proof. Let $\boldsymbol{y}_0 \in \mathcal{R}_0$. According to Definition 4, we have that

$$\forall i \in \mathbb{N}.\ \forall \pi \in \Pi.\ \forall j \in \{1,\ldots,n_0\}.\ h_{0,j}(\boldsymbol{x}^{\pi}_{\boldsymbol{y}_0}(i)) \leq 0 \tag{4}$$

holds. Therefore, $V(\boldsymbol{y}_0) \leq 0$ and thus $\boldsymbol{y}_0 \in \{\boldsymbol{x}_0 \mid V(\boldsymbol{x}_0) \leq 0\}$.

On the other side, if $\boldsymbol{y}_0 \in \{\boldsymbol{x}_0 \in \mathbb{R}^n \mid V(\boldsymbol{x}_0) \leq 0\}$, then $V(\boldsymbol{y}_0) \leq 0$, implying that (4) holds. Therefore, $\boldsymbol{y}_0 \in \mathcal{R}_0$.

This concludes that $\mathcal{R}_0 = \{\boldsymbol{x}_0 \in \mathbb{R}^n \mid V(\boldsymbol{x}_0) \leq 0\}$.

From Theorem 1, the maximal robust nontermination set $\mathcal{R}_0$ could be constructed by computing $V(\boldsymbol{x}_0)$, which satisfies the dynamic programming principle as presented in Lemma 1.

Lemma 1. *For $\forall \boldsymbol{x}_0 \in \mathbb{R}^n$ and $\forall l \in \mathbb{N}$, we have:*

$$V(\boldsymbol{x}_0) = \sup_{\pi \in \Pi} \max \Big\{V(\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(l)), \sup_{i \in [0,l) \cap \mathbb{N}} \max_{j \in \{1,\ldots,n_0\}} h_{0,j}(\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(i))\Big\}. \tag{5}$$

Proof. Let

$$W(l, \boldsymbol{x}_0) := \sup_{\pi \in \Pi} \max \Big\{V(\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(l)), \sup_{i \in [0,l) \cap \mathbb{N}} \max_{j \in \{1,\ldots,n_0\}} h_{0,j}(\boldsymbol{x}^{\pi}_{\boldsymbol{x}_0}(i))\Big\}. \tag{6}$$

We will prove that for $\epsilon > 0$, $|W(l, \boldsymbol{x}_0) - V(\boldsymbol{x}_0)| < \epsilon$.

According to the definition of $V(\boldsymbol{x}_0)$, i.e. (3), for any ϵ_1, there exists an input policy π' such that

$$V(\boldsymbol{x}_0) \leq \sup_{i \in \mathbb{N}} \max_{j \in \{1,\ldots,n_0\}} \{h_{0,j}(\boldsymbol{x}^{\pi'}_{\boldsymbol{x}_0}(i))\} + \epsilon_1.$$

We then introduce two infinite uncertain input policies π_1 and π_2 such that $\pi_1 = \{\boldsymbol{\pi}_1(i), i \in \mathbb{N}\}$ with $\boldsymbol{\pi}_1(j) = \boldsymbol{\pi}'(j)$ for $j = 0, \dots, l-1$ and $\pi_2 = \{\boldsymbol{\pi}_2(i), i \in \mathbb{N}\}$ with $\boldsymbol{\pi}(j) = \boldsymbol{\pi}'(j+l)$ $\forall j \in \mathbb{N}$. Now, let $\boldsymbol{y} \in \boldsymbol{x}_{\boldsymbol{x}_0}^{\pi_1}(l)$, then we obtain that

$$\begin{aligned}
&W(l, \boldsymbol{x}_0) \geq \max\big\{V(\boldsymbol{y}), \sup_{i \in [0,l) \cap \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} h_{0,j}(\boldsymbol{x}_{\boldsymbol{y}}^{\pi_1}(i))\big\} \\
&\geq \max\big\{ \sup_{i \in [l,+\infty) \cap \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi_2}(i-l))\}, \sup_{i \in [0,l) \cap \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi_1}(i))\}\big\} \\
&= \max\big\{ \sup_{i \in [l,+\infty) \cap \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi'}(i))\}, \sup_{i \in [0,l) \cap \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi'}(i))\}\big\} \\
&= \sup_{i \in \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi'}(i))\} \\
&\geq V(\boldsymbol{x}_0) - \epsilon_1.
\end{aligned}$$

Therefore,

$$V(\boldsymbol{x}_0) \leq W(l, \boldsymbol{x}_0) + \epsilon_1. \tag{7}$$

On the other hand, for any $\epsilon_1 > 0$, there exists a $\pi_1 \in \Pi$ such that $W(l, \boldsymbol{x}_0) \leq \max\big\{V(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi_1}(l)), \sup_{i \in [0,l) \cap \mathbb{N}} \max_{j \in \{1,\dots,n_0\}}\{h_{0,j}(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi_1}(i))\}\big\} + \epsilon_1$, by the definition of $W(l, \boldsymbol{x}_0)$. Also, by the definition of $V(\boldsymbol{x}_0)$, i.e. (3), for any $\epsilon_1 > 0$, there exists a π_2 such that

$$V(\boldsymbol{y}) \leq \sup_{i \in \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{y}}^{\pi_2}(i))\} + \epsilon_1,$$

where $\boldsymbol{y} = \boldsymbol{x}_{\boldsymbol{x}_0}^{\pi_1}(l)$. We define $\pi \in \Pi$ such that $\boldsymbol{\pi}(i) = \boldsymbol{\pi}_1(i)$ for $i = 0, \dots, l-1$ and $\boldsymbol{\pi}(i+l) = \boldsymbol{\pi}_2(i)$ for $\forall i \in \mathbb{N}$. Then, it follows

$$\begin{aligned}
W(l, \boldsymbol{x}_0) &\leq 2\epsilon_1 + \max\{ \sup_{i \in \mathbb{N} \cap [l,\infty)} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{y}}^{\pi_2}(i-l))\}, \\
&\qquad \sup_{i \in [0,l) \cap \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi_1}(i))\}\} \\
&\leq \sup_{i \in [0,+\infty) \cap \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}_{\boldsymbol{x}_0}^{\pi}(i))\} + 2\epsilon_1 \\
&\leq V(\boldsymbol{x}_0) + 2\epsilon_1.
\end{aligned} \tag{8}$$

Combining (7) and (8), we finally have $|V(\boldsymbol{x}_0) - W(l, \boldsymbol{x}_0)| \leq \epsilon = 2\epsilon_1$, implying that $V(\boldsymbol{x}_0) = W(l, \boldsymbol{x}_0)$ since ϵ_1 is arbitrary. This completes the proof.

Based on Lemma 1 stating that the value function $V(\boldsymbol{x}_0)$ complies with the dynamic programming principle (5), we derive a central equation of this paper, which is formulated formally in Theorem 2.

Theorem 2. *The value function $V(\boldsymbol{x}_0) : \mathbb{R}^n \mapsto \mathbb{R}$ in (3) is a solution to the equation*

$$\min\big\{ \inf_{\boldsymbol{d} \in D} (V(\boldsymbol{x}_0) - V(\boldsymbol{f}(\boldsymbol{x}_0, \boldsymbol{d}))), V(\boldsymbol{x}_0) - \max_{j \in \{1,\dots,n_0\}} h_{0,j}(\boldsymbol{x}_0)\big\} = 0. \tag{9}$$

Proof. It is evident that (9) is derived from (5) when $l = 1$.

According to Theorem 2, we conclude that *if there does not exist a solution to* (9)*, the robust nontermination set* $\mathcal{R}_0$ *is empty.* Moreover, according to Theorem 2, $V(\boldsymbol{x}_0)$ as defined in (3) is a solution to (9). Note that the solution to (9) may be not unique, and we do not go deeper into this matter in this paper. However, any solution to (9) forms an inner-approximation of the maximal robust nontermination set, as stated in Corollary 1.

Corollary 1. *For any function* $u(\boldsymbol{x}_0) : \mathbb{R}^n \mapsto \mathbb{R}$ *satisfying* (9)*,* $\{\boldsymbol{x}_0 \in \mathbb{R}^n \mid u(\boldsymbol{x}_0) \leq 0\}$ *is an inner-approximation of the maximal robust nontermination set* $\mathcal{R}_0$*, i.e.* $\{\boldsymbol{x}_0 \in \mathbb{R}^n \mid u(\boldsymbol{x}_0) \leq 0\} \subset \mathcal{R}_0$.

Proof. Let $u(\boldsymbol{x}_0) : \mathbb{R}^n \mapsto \mathbb{R}$ be a solution to (9). It is evident that $u(\boldsymbol{x}_0)$ satisfies the constraints:

$$\begin{cases} u(\boldsymbol{x}_0) - u(\boldsymbol{f}(\boldsymbol{x}_0, \boldsymbol{d})) \geq 0, \forall \boldsymbol{x}_0 \in \mathbb{R}^n, \forall \boldsymbol{d} \in D, \\ u(\boldsymbol{x}_0) - h_{0,j}(\boldsymbol{x}_0) \geq 0, \qquad \forall \boldsymbol{x}_0 \in \mathbb{R}^n, \forall j \in \{1, \dots, n_0\} \end{cases} \tag{10}$$

Assume $\boldsymbol{x}'_0 \in \{\boldsymbol{x}_0 \mid u(\boldsymbol{x}_0) \leq 0\}$. According to (10), we have that for $\forall \pi \in \Pi$, $\forall l \in \mathbb{N}$ and $\forall j \in \{1, \dots, n_0\}$,

$$\begin{cases} u(\boldsymbol{x}^{\pi}_{\boldsymbol{x}'_0}(l+1)) & \leq u(\boldsymbol{x}^{\pi}_{\boldsymbol{x}'_0}(l)) \leq u(\boldsymbol{x}'_0) \\ h_{0,j}(\boldsymbol{x}^{\pi}_{\boldsymbol{x}'_0}(l)) & \leq u(\boldsymbol{x}^{\pi}_{\boldsymbol{x}'_0}(l)) \leq u(\boldsymbol{x}'_0) \end{cases}. \tag{11}$$

Therefore, $\sup_{l \in \mathbb{N}} \max_{j \in \{1,\dots,n_0\}} \{h_{0,j}(\boldsymbol{x}^{\pi}_{\boldsymbol{x}'_0}(l))\} \leq u(\boldsymbol{x}'_0) \leq 0$, implying that $\boldsymbol{x}'_0 \in \mathcal{R}_0$. Thus, $\{\boldsymbol{x}_0 \in \mathbb{R}^n \mid u(\boldsymbol{x}_0) \leq 0\} \subset \mathcal{R}_0$.

From Corollary 1, it is clear that an approximation of $\mathcal{R}_0$ from inside, i.e. a robust nontermination set, is able to be constructed by addressing (9). The solution to (9) could be addressed by grid-based numerical methods such as level set methods [12,32], which are a popular method for interface capturing. Such grid-based methods are prohibitive for systems of dimension greater than four without relying upon specific system structure. Besides, we observe that a robust nontermination set could be searched by solving (10) rather than (9). In the subsection that follows we relax (10) as a sum-of-squares decomposition problem in a semidefinite programming formulation when in Program 1, $\boldsymbol{f}_i$s are polynomials over $\boldsymbol{x}$ and $\boldsymbol{d}$, state and uncertain input constraints, i.e. X_js and Ds, are restricted to basic semi-algebraic sets.

3.2 Semi-definite Programming Implementation

In practice, it is non-trivial to obtain a solution $V(\boldsymbol{x}_0)$ to (2), and thus non-trivial to gain $\mathcal{R}_0$. In this subsection, thanks to (10) and Corollary 1, we present a semi-definite programming based method to solve (9) approximately and construct a robust invariant Ω as presented in Definition 4 when Assumption 1 holds.

Assumption 1. $\boldsymbol{f}_i$*,* $i = 1, \dots, k$*, is polynomial over* $\boldsymbol{x}$ *and* $\boldsymbol{d}$*,* X_j *and* D*,* $j = 0, \dots, k$*, are restricted to basic semi-algebraic sets in* Program 1.

Firstly, (10) has indicator functions on the expression $u(\boldsymbol{x}_0) - u(\boldsymbol{f}(\boldsymbol{x}_0, \boldsymbol{d}))$, which is beyond the capability of the solvers we use. We would like to obtain a constraint by removing indicators according to Lemma 2.

Lemma 2 **([8]).** *Suppose* $\boldsymbol{f}'(\boldsymbol{x}) = 1_{F_1} \cdot \boldsymbol{f}'_1(\boldsymbol{x}) + \cdots + 1_{F_{k'}} \cdot \boldsymbol{f}'_{k'}(\boldsymbol{x})$ *and* $\boldsymbol{g}'(\boldsymbol{x}) = 1_{G_1} \cdot \boldsymbol{g}'_1(\boldsymbol{x}) + \cdots + 1_{G_{l'}} \cdot \boldsymbol{g}'_{l'}(\boldsymbol{x})$, *where* $\boldsymbol{x} \in \mathbb{R}^n$, $k', l' \in \mathbb{N}$, *and* $F_i, G_j \subseteq \mathbb{R}^n$, $i = 1, \ldots, k'$, $j = 1, \ldots, l'$. *Also,* $F_1, \ldots, F_{k'}$ *and* $G_1, \ldots, G_{l'}$ *are respectively disjoint. Then,* $\boldsymbol{f}' \leq \boldsymbol{g}'$ *if and only if (pointwise)*

$$\begin{aligned} &\bigwedge_{i=1}^{k'} \bigwedge_{j=1}^{l'} \left[F_i \wedge G_j \Rightarrow \boldsymbol{f}'_i \leq \boldsymbol{g}'_j\right] \wedge \\ &\qquad \bigwedge_{i=1}^{k'} \left[F_i \wedge \left(\bigwedge_{j=1}^{l'} \neg G_j\right) \Rightarrow \boldsymbol{f}'_i \leq 0\right] \wedge \\ &\qquad\qquad \bigwedge_{j=1}^{l'} \left[\left(\bigwedge_{i=1}^{k'} \neg F_i\right) \wedge G_j \Rightarrow 0 \leq \boldsymbol{g}'_j\right]. \end{aligned} \tag{12}$$

Consequently, according to Lemma 2, the equivalent constraint without indicator functions of (10) is equivalently formulated below:

$$\begin{aligned} &\bigwedge_{i=1}^{k} \left[\forall \boldsymbol{d} \in D.\ \forall \boldsymbol{x}_0 \in X_i.\ u(\boldsymbol{x}_0) - u(\boldsymbol{f}_i(\boldsymbol{x}_0, \boldsymbol{d})) \geq 0\right] \wedge \\ &\bigwedge_{j=1}^{n_0} \left[\forall \boldsymbol{x}_0 \in \mathbb{R}^n.\ u(\boldsymbol{x}_0) - h_{0,j}(\boldsymbol{x}_0) \geq 0\right]. \end{aligned} \tag{13}$$

Before encoding (13) in sum-of-squares programming formulation, we denote the set of sum of squares polynomials over variables $\boldsymbol{y}$ by $\texttt{SOS}(\boldsymbol{y})$, i.e.

$$\texttt{SOS}(\boldsymbol{y}) := \{p \in \mathbb{R}[\boldsymbol{y}] \mid p = \sum_{i=1}^{r} q_i^2, q_i \in \mathbb{R}[\boldsymbol{y}], i = 1, \ldots, r\}.$$

Besides, we define the set $\Omega(X_0)$ of states being reachable from the set X_0 within one step computation, i.e.,

$$\Omega(X_0) := \{\boldsymbol{x} \mid \boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x}_0, \boldsymbol{d}), \boldsymbol{x}_0 \in X_0, \boldsymbol{d} \in D\} \cup X_0, \tag{14}$$

which can be obtained by semi-definite programming or linear programming methods as in [24,31]. Herein, we assume that it was already given. Consequently, when Assumption 1 holds and $u(\boldsymbol{x})$ in (13) is constrained to polynomial type and is restricted in a ball $B = \{\boldsymbol{x} \mid h(\boldsymbol{x}) \geq 0\}$, where $h(\boldsymbol{x}) = R - \sum_{i=1}^{n} x_i^2$ and $\Omega(X_0) \subseteq B$, (13) is relaxed as the following sum-of-squares programming problem:

$$\begin{aligned}
&\min_{u, s^{X_i}_{i,l_1}, s^{D}_{i,l_2}, s_{i,l}, s'_{1,j}} \quad \boldsymbol{c}' \cdot \boldsymbol{w} \\
&u(\boldsymbol{x}) - u(\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{d})) + \sum_{l_1=1}^{n_i} s^{X_i}_{i,l_1} h_{i,l_1}(\boldsymbol{x}) + \sum_{l_2=1}^{n_{k+1}} s^{D}_{i,l_2} h_{k+1,l}(\boldsymbol{d}) - s_{i,1} h(\boldsymbol{x}) \in \mathtt{SOS}(\boldsymbol{x},\boldsymbol{d}), \\
&u(\boldsymbol{x}) - h_{0,j}(\boldsymbol{x}) - s'_{1,j} h(\boldsymbol{x}) \in \mathtt{SOS}(\boldsymbol{x}), \\
&i = 1,\dots,k; j = 1,\dots,n_0,
\end{aligned} \tag{15}$$

where $\boldsymbol{c}' \cdot \boldsymbol{w} = \int_B u d\mu(\boldsymbol{x})$, $\boldsymbol{w}$ is the vector of the moments of the Lebesgue measure over B indexded in the same basis in which the polynomial $u(\boldsymbol{x}) \in \mathbb{R}_d[\boldsymbol{x}]$ with coefficients $\boldsymbol{c}$ is expressed, $s^{X_i}_{i,l_1}, s^{D}_{i,l_2}, s_{i,1} \in \mathtt{SOS}(\boldsymbol{x},\boldsymbol{d})$, $i = 1,\dots,k$, $l_1 = 1,\dots,n_i$, $l_2 = 1,\dots,n_{k+1}$, $s'_{1,j} \in \mathtt{SOS}(\boldsymbol{x})$, $j = 1,\dots,n_0$, are sum-of-squares polynomials of appropriate degree. The constraints that polynomials are sum-of-squares can be written explicitly as linear matrix inequalities, and the objective is linear in the coefficients of the polynomial $u(\boldsymbol{x})$; therefore problem (15) is reformulated as an semi-definite program, which falls within the convex programming framework and can be solved via interior-points method in polynomial time (e.g., [52]). Note that the objective of (15) facilitate the gain of the less conservative robust nontermination set.

The implementation based on the sum-of-squares program (15) is sound but incomplete. Its soundness is presented in Theorem 3.

Theorem 3 (Soundness). *Let $u(\boldsymbol{x}) \in \mathbb{R}_d[\boldsymbol{x}]$ be solution to* (15)*, then $\{\boldsymbol{x} \in B \mid u(\boldsymbol{x}) \le 0\}$ is an inner-approximation of $\mathcal{R}_0$, i.e., every possible run of* Program 1 *starting from a state in $\{\boldsymbol{x} \in B \mid u(\boldsymbol{x}) \le 0\}$ does not terminate.*

Proof. Since $u(\boldsymbol{x})$ satisfies the constraint in (15), we obtain that $u(\boldsymbol{x})$ satisfies according to $\mathcal{S}-$ procedure in [5]:

$$\bigwedge_{i=1}^{k} \left[\forall \boldsymbol{d} \in D.\ \forall \boldsymbol{x} \in X_i \cap B.\ u(\boldsymbol{x}) - u(\boldsymbol{f}_i(\boldsymbol{x},\boldsymbol{d})) \ge 0\right] \wedge \tag{16}$$

$$\bigwedge_{j=1}^{n_0} \left[\forall \boldsymbol{x} \in B.\ u(\boldsymbol{x}) - h_{0,j}(\boldsymbol{x}) \ge 0\right]. \tag{17}$$

Due to (16) and the fact that $\cup_{i=1}^{k} X_i = \mathbb{R}^n$, we obtain that for $\forall \boldsymbol{x}_0 \in \{\boldsymbol{x} \in B \mid u(\boldsymbol{x}) \le 0\}$, $\exists i \in \{1,\dots,k\}$. $\forall \boldsymbol{d} \in D$. $u(\boldsymbol{x}_0) - u(\boldsymbol{f}_i(\boldsymbol{x}_0,\boldsymbol{d})) \ge 0$, implying that

$$u(\boldsymbol{x}_0) - u(\boldsymbol{f}(\boldsymbol{x}_0,\boldsymbol{d})) \ge 0, \forall \boldsymbol{d} \in D. \tag{18}$$

Assume that there exist an initial state $\boldsymbol{y}_0 \in \{\boldsymbol{x} \in B \mid u(\boldsymbol{x}) \le 0\}$ and an input policy π' such that $\boldsymbol{x}^{\pi'}_{\boldsymbol{y}_0}(l) \in X_0$ does not hold for $\forall l \in \mathbb{N}$. Due to the fact that (17) holds, we have the conclusion that $\{\boldsymbol{x} \in B \mid u(\boldsymbol{x}) \le 0\} \subset X_0$ and thus $\boldsymbol{y}_0 \in X_0$. Let $l_0 \in \mathbb{N}$ be the first time making $\boldsymbol{x}^{\pi'}_{\boldsymbol{y}_0}(l)$ violate the constraint X_0, i.e., $\boldsymbol{x}^{\pi'}_{\boldsymbol{y}_0}(l_0) \notin X_0$ and $\boldsymbol{x}^{\pi'}_{\boldsymbol{y}_0}(l) \in X_0$ for $l = 0,\dots,l_0-1$. Also, since $\Omega(X_0) \subset B$, (18) and (17), where $\Omega(X_0)$ is defined in (14), we derive that $\boldsymbol{x}^{\pi'}_{\boldsymbol{y}_0}(l_0-1) \in \{\boldsymbol{x} \in B \mid u(\boldsymbol{x}) \le 0\}$ and $u(\boldsymbol{x}^{\pi'}_{\boldsymbol{y}_0}(l_0)) > 0$, which contradicts (18).

Thus, every possible run of Program 1 initialized in $\{\boldsymbol{x} \in B \mid u(\boldsymbol{x}) \leq 0\}$ will live in $\{\boldsymbol{x} \in B \mid u(\boldsymbol{x}) \leq 0\}$ forever while respecting X_0.

Therefore, the conclusion in Theorem 3 is justified.

4 Experiments

In this section we evaluate the performance of our method built upon the semi-definite program (15). Examples 2 and 3 are constructed to illustrate the soundness of our method. Example 4 is used to evaluate the scalability of our method in dealing with Program 1. The parameters that control the performance of our approach in applying (15) to these three examples are presented in Table 1. All computations were performed on an i7-7500U 2.70GHz CPU with 32GB RAM running Windows 10. For numerical implementation, we formulate the sum of squares problem (15) using the MATLAB package YALMIP[1] [29] and use Mosek[2] [33] as a semi-definite programming solver.

Table 1. Parameters and performance of our implementations on the examples presented in this section. $d_u, d_{s^{X_i}_{i,l_1}}, d_{s^D_{i,l_2}}, d_{s_{i,l}}, d_{s'_{1,j}}$: the degree of the polynomials $u, s^{X_i}_{i,l_1}, s^D_{i,l_2}, s_{i,l}, s'_{1,j}$ in (15), respectively, $i = 1, \ldots, k$, $l_1 = 1, \ldots, n_i$, $l_2 = 1, \ldots, n_{k+1}$, $j = 1, \ldots, n_0$; *Time*: computation times (seconds).

Ex.	d_h	$d_{s^{X_i}_{i,l_1}}$	$d_{s^D_{i,l_2}}$	$d_{s_{i,l}}$	$d_{s'_{1,j}}$	*Time*
1	14	14	14	14	14	11.30
1	16	16	16	16	16	28.59
2	6	12	12	12	6	9.06
2	8	16	16	16	8	65.22
2	10	20	20	20	10	123.95
2	12	24	24	24	12	623.95
4	4	4	4	4	4	58.56
4	5	4	4	4	4	60.02

Example 2. This simple example is mainly constructed to illustrate the difference between Program 1 taking uncertain inputs into account and free of disturbances. In both cases, Program 1 is composed of a single loop without switch-case type in loop body, i.e. $k = 1$ and $X_1 = \mathbb{R}^2$.

In case that $\boldsymbol{f}_1(x, y) = (0.4x + 0.6y; dx + 0.9y)$, $X_0 = \{(x, y) \mid x^2 + y^2 - 1 \leq 0\}$ and $D = \{d \mid d^2 - 0.01 \leq 0\}$ in Program 1, the inner-approximations of the

[1] It can be downloaded from https://yalmip.github.io/.

[2] For academic use, the software Mosek can be obtained free from https://www.mosek.com/.

maximal robust nontermination set $\mathcal{R}_0$ are illustrated in Fig. 2(Left) when $d_u = 10$ and $d_u = 12$. By visualizing the results in Fig. 2, the inner-approximation obtained when $d_u = 12$ does not improve the one when $d_u = 10$ a lot. Although there is a gap between the inner-approximations obtained via our method and the set $\mathcal{R}_0$, it is not big.

In the ideal implementation of Program 1, that is, d in the loop body is a fixed nominal value, there will exists some initial conditions such that Program 1 in the real implementation may violate the constraint set X_0, i.e. Program 1 may terminate. We use $d = 0$ as an instance to illustrate such situation. The difference between termination sets is visualized in Fig. 2(Right). The robust nontermination set in case of $d \in [-0.1, 0.1]$ is smaller than the nontermination set when $d = 0$. Note that from Fig. 3, we observe that the inner-approximation obtained by our method when $d_u = 10$ can approximate $\mathcal{R}_0$ very well.

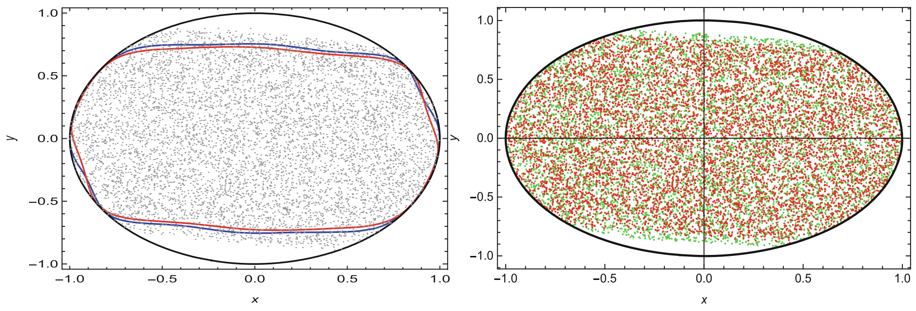


Fig. 2. Computed robust nontermination sets for Example 2. Left: (Blue and Red curves – the boundaries of the computed robust nontermination set $\mathcal{R}_0$ when $d_u = 10$ and $d_u = 12$, respectively; Gray points – the approximated robust nontermination set via numerical simulation techniques; Black curve – the boundary of X_0.) Right: (Green and red points – the approximated (robust) nontermination sets via numerical simulation techniques for Program 1 without and with disturbance inputs, respectively; Black curve – the boundary of X_0.) (Color figure online)

Example 3. In this example we consider Program 1 with switch-case type in the loop body, where $\boldsymbol{f}_1(x, y) = (x; (0.5 + d)x - 0.1y)$, $\boldsymbol{f}_2(x, y) = (y; 0.2x - (0.1 + d)y + y^2)$, $X_0 = \{(x, y) \mid x^2 + y^2 - 0.8 \leq 0\}$, $X_1 = \{(x, y) \mid 1 - (x - 1)^2 - y^2 \geq 0\}$, $X_2 = \{(x, y) \mid -1 + (x - 1)^2 + y^2 < 0\}$ and $D = \{d \mid d^2 - 0.01 \leq 0\}$. The inner-approximations computed by solving (15) when $d_u = 8, 10$ and 12 respectively are illustrated in Fig. 4. By comparing these results, we observe that polynomials of higher degree facilitate the gain of less conservative estimation of the set $\mathcal{R}_0$.

Example 4. In this example, we consider Program 1 with seven variables $\boldsymbol{x} = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ and illustrate the scalability of our approach. In Program 1, $\boldsymbol{f}_1(\boldsymbol{x}) = ((0.5 + d)x_1; 0.8x_2; 0.6x_3 + 0.1x_6; x_4; 0.8x_5; 0.1x_2 + x_6; 0.2x_2 +$

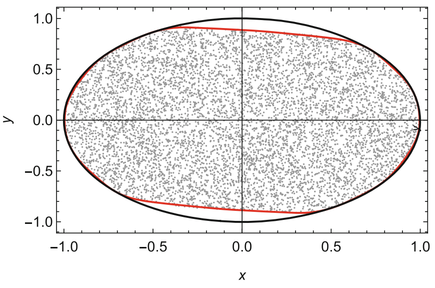


Fig. 3. Nontermination set estimation for Example 2. (Black and Red curves: the boundaries of X_0 and the computed robust nontermination set $\mathcal{R}_0$ when $d_u = 16$, respectively; Gray points – the approximated robust nontermination set via numerical simulation techniques.) (Color figure online)

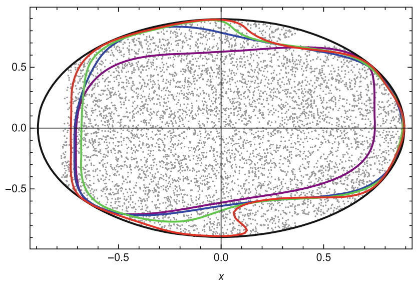


Fig. 4. Robust nontermination sets for Example 3. Black, Purple, Blue, Green and Red curves: the boundaries of X_0 and the computed robust nontermination sets $\mathcal{R}_0$ when $d_u = 6, 8, 10, 12$, respectively; Gray points – the approximated robust nontermination set via numerical simulation techniques. (Color figure online)

$0.6x_7)$;, $\boldsymbol{f}_2(\boldsymbol{x}) = (0.5x_1{+}0.1x_6; (0.5{+}d)x_2; x_3; 0.1x_1{+}0.4x_4; 0.2x_1{+}x_5; x_6; 0.1x_1{+}x_7)$, $X_0 = \{\boldsymbol{x} \mid \sum_{i=1}^{7} x_i^2 - 1 \geq 0\}$, $X_1 = \{\boldsymbol{x} \mid x_1 + x_2 + x_3 - x_4 - x_5 - x_6 - x_7 \geq 0\}$, $X_2 = \{(x, y) \mid x_1 + x_2 + x_3 - x_4 - x_5 - x_6 - x_7 < 0\}$ and $D = \{d \mid d^2 - 0.01 \leq 0\}$. From the computation times listed Table 1, we conclude that although the computation time increases with the number of variables increasing, our method may deal with problems with many variables, especially for the cases that the robust nontermination set formed by a polynomial of low degree fulfills certain needs in real applications. Note that numerical simulation techniques suffers from the curse of dimensionality and thus can not apply to this example since this example has seven variables, we just illustrate the results computed by our method based on (15) in Fig. 5.

5 Related Work

Methods for proving nontermination of programs have recently been studied actively. [16] uses a characterization of nontermination by recurrence sets of states that is visited infinitely often along the path. A recurrence set exists iff a program is non-terminating. To find recurrence sets they provide a method based on constraint solving. Their method is only applicable to programs with linear integer arithmetic and does not support non-determinism and is implemented in the tool TNT. [7] proposes a method combining closed recurrence sets with counterexample-guided underapproximation for disproving termination. This method, implemented in the tool T2, relies heavily on suitable safety provers for the class of programs of interest, thus rendering an application of

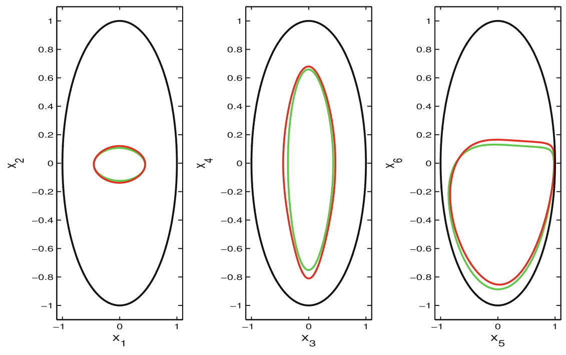


Fig. 5. Computed robust nontermination sets for Example 4. (Black, Red and Green curves – the boundaries of X_0 and the cross-sections (from left to right: $x_3 = x_4 = x_5 = x_6 = x_7 = 0$, $x_1 = x_2 = x_5 = x_6 = x_7 = 0$ and $x_1 = x_2 = x_3 = x_4 = x_7 = 0$) of the computed robust nontermination sets $\mathcal{R}_0$ when $d_u = 5$ and $d_u = 4$, respectively.) (Color figure online)

their method to nonlinear programs difficult. Further, [9] introduces live abstractions combing with closed recurrence sets to disprove termination. However, this method, implemented in the tool ANANT, is only suitable for disproving nontermination in the form of lasso for programs of finite control-flow graphs.

There are also some approaches exploiting theorem-proving techniques to prove nontermination, e.g., [53] presents a method for disproving nontermination of Java programs based on theorem proving and generation of invariants. This method is implemented in INVEL, which is restricted to deterministic programs with unbounded integers and single loops. APROVE [14] uses SMT solving to prove nontermination of Java programs [6]. The application of this method requires either singleton recurrence sets or loop conditions being recurrence sets in the programs of interest. [23] disproves termination based on MaxSMT-based invariant generation, which is implemented in the tool CPPINV. This method is limited to linear arithmetic as well.

Besides, TREX [17] integrates existing non-termination proving approaches to develop compositional analysis algorithms for detecting non-termination in multithreaded programs. Different from the method in TREX targeting sequential code, [1] presents a nontermination proving technique for multi-threaded programs via a reduction to nontermination reasoning for sequential programs. [27] investigates the termination problems of multi-path polynomial programs with equational loop guards and discovering nonterminating inputs for such programs. It shows that the set of all strong non-terminating inputs and weak non-terminating inputs both correspond to the real varieties of certain polynomial ideals. Recently, [22] proposes a method combining higher-order model checking with predicate abstraction and CEGAR for disproving nontermination of higher-order functional programs. This method reduces the problem of disproving non-termination to the problem of checking a certain branching property of an abstract program, which can be solved by higher-order model checking.

Please refer to [10, 55] for detailed surveys on termination and nontermination analysis of programs.

As opposed to above works without considering robust nontermination, by taking disturbances such as round-off errors in performing numerical implementation of computer programs into account, this paper propose a systematic approach for proving robust nontermination of a class of computer programs, which are composed of a single loop with a possibly complicated switch-case type loop body and encountered often in current embedded systems. The problem of robust conditional nontermination is reduced to a problem of solving a single equation derived via dynamic programming principle, and semi-definite programs could be employed to solve such optimal control problem efficiently in some situations.

The underlying idea in this work is in sprit analogous to that in [40], which is pioneer in proposing a systematic framework to conduct verification of numerical software based on Lyapunov invariance in control theroy. Our method for conducting (robust) verification of numerical software falls within the framework proposed in [40]. The primary contribution of our work is that we systematically investigate a class of computer programs and reduce the nontermination problem for such computer programs to a mathematical equation, thus resulting in an efficient nontermination verification method, as indicated in Introduction Sect. 1.

6 Conclusion and Future Work

In this paper we presented a system-theoretic framework to numerical software analysis and considered the problem of conditional robust non-termination analysis for a class of computer programs composed of a single loop with a possibly complicated switch-case type loop body, which is encountered often in real-time embedded systems. The maximal robust nontermination set of initial configurations in our method was characterized by a solution to a mathematical equation. Although it is non-trivial to solve gained equation, in the case of polynomial assignments in the loop body and basic semi-algebraic sets in Program 1, the equation could be relaxed as a semi-definite program, which falls within the convex programming framework and can be solved efficiently via interior point methods. Finally, we have reported experiments with encouraging results to demonstrate the merits of our method.

However, there are a lot of works remaining to be done. For instance, the semi-definite programming solver is implemented with floating point computations, we have no absolute guarantee on the results it provides. In future work, we need a sound and efficient verification procedure such as that presented in [26, 35, 41, 46] that is able to check the result from the solver and help us decide whether the result is qualitatively correct. Besides, the presented work can be extended in several directions, these include robust nontermination analysis for computer programs with nested loops and robust invariant generations with or without constraints [20, 28, 42].

References

1. Atig, M.F., Bouajjani, A., Emmi, M., Lal, A.: Detecting fair non-termination in multithreaded programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 210–226. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_19
2. Blanchini, F., Miani, S.: Set-Theoretic Methods in Control. Springer, Boston (2008). https://doi.org/10.1007/978-0-8176-4606-6
3. Borralleras, C., Brockschmidt, M., Larraz, D., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving termination through conditional termination. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 99–117. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_6
4. Bourbaki, N.: General Topology: Chapters 1–4, vol. 18. Springer, Heidelberg (2013)
5. Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: Linear Matrix Inequalities in System and Control Theory. SIAM, Philadelphia (1994)
6. Brockschmidt, M., Ströder, T., Otto, C., Giesl, J.: Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 123–141. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31762-0_9
7. Chen, H.-Y., Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Proving nontermination via safety. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 156–171. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_11
8. Chen, Y.-F., Hong, C.-D., Wang, B.-Y., Zhang, L.: Counterexample-guided polynomial loop invariant generation by lagrange interpolation. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 658–674. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_44
9. Cook, B., Fuhs, C., Nimkar, K., O'Hearn, P.: Disproving termination with overapproximation. In: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design, pp. 67–74. FMCAD Inc. (2014)
10. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM **54**(5), 88–98 (2011)
11. Cousot, P., Cousot, R.: A gentle introduction to formal verification of computer systems by abstract interpretation (2010)
12. Fedkiw, S.O.R., Osher, S.: Level set methods and dynamic implicit surfaces. Surfaces **44**, 77 (2002)
13. Floyd, R.W.: Assigning meanings to programs. Math. Aspects Comput. Sci. **19**(19–32), 1 (1967)
14. Giesl, J., et al.: Proving termination of programs automatically with AProVE. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 184–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_13
15. Giesl, P., Hafstein, S.: Review on computational methods for Lyapunov functions. Discrete Contin. Dyn. Syst.-Ser. B **20**(8), 2291–2331 (2015)
16. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving non-termination. ACM Sigplan Not. **43**(1), 147–158 (2008)
17. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for termination. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 304–319. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15769-1_19

18. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
19. Jarvis-Wloszek, Z.W.: Lyapunov based analysis and controller synthesis for polynomial systems using sum-of-squares optimization. Ph.D. thesis, University of California, Berkeley (2003)
20. Kapur, D.: Automatically generating loop invariants using quantifier elimination. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2006)
21. Kouramas, K.I., Rakovic, S.V., Kerrigan, E.C., Allwright, J., Mayne, D.Q.: On the minimal robust positively invariant set for linear difference inclusions. In: 44th IEEE Conference on Decision and Control, 2005 and 2005 European Control Conference, CDC-ECC 2005, pp. 2296–2301. IEEE (2005)
22. Kuwahara, T., Sato, R., Unno, H., Kobayashi, N.: Predicate abstraction and CEGAR for disproving termination of higher-order functional programs. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 287–303. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21668-3_17
23. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using max-SMT. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 779–796. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_52
24. Lasserre, J.B.: Tractable approximations of sets defined with quantifiers. Math. Programm. **151**(2), 507–527 (2015)
25. Li, Y.: Witness to non-termination of linear programs. Theor. Comput. Sci. **681**, 75–100 (2017)
26. Lin, W., Wu, M., Yang, Z., Zeng, Z.: Exact safety verification of hybrid systems using sums-of-squares representation. Sci. China Inf. Sci. **57**(5), 1–13 (2014)
27. Liu, J., Xu, M., Zhan, N., Zhao, H.: Discovering non-terminating inputs for multi-path polynomial programs. J. Syst. Sci. Complex. **27**(6), 1286–1304 (2014)
28. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: Proceedings of the Ninth ACM International Conference on Embedded Software, pp. 97–106. ACM (2011)
29. Lofberg, J.: YALMIP: a toolbox for modeling and optimization in MATLAB. In: 2004 IEEE International Symposium on Computer Aided Control Systems Design, pp. 284–289. IEEE (2004)
30. Luk, C.K., Chesi, G.: On the estimation of the domain of attraction for discrete-time switched and hybrid nonlinear systems. Int. J. Syst. Sci. **46**(15), 2781–2787 (2015)
31. Magron, V., Garoche, P.-L., Henrion, D., Thirioux, X.: Semidefinite approximations of reachable sets for discrete-time polynomial systems. arXiv preprint arXiv:1703.05085 (2017)
32. Mitchell, I.M., Bayen, A.M., Tomlin, C.J.: A time-dependent Hamilton-Jacobi formulation of reachable sets for continuous dynamic games. IEEE Trans. Autom. Control **50**(7), 947–957 (2005)
33. Mosek, A.: The MOSEK optimization toolbox for MATLAB manual. Version 7.1 (Revision 28), p. 17 (2015)
34. Naur, P.: Proof of algorithms by general snapshots. BIT Numer. Math. **6**(4), 310–316 (1966)
35. Platzer, A., Quesel, J.-D., Rümmer, P.: Real world verification. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 485–501. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02959-2_35

36. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24743-2_32
37. Prajna, S., Jadbabaie, A., Pappas, G.J.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Transa. Autom. Control **52**(8), 1415–1428 (2007)
38. Rakovic, S.V., Kerrigan, E.C., Kouramas, K.I., Mayne, D.Q.: Invariant approximations of the minimal robust positively invariant set. IEEE Trans. Autom. Control **50**(3), 406–410 (2005)
39. Rebiha, R., Matringe, N., Moura, A.V.: Generating asymptotically non-terminating initial values for linear programs. arXiv preprint arXiv:1407.4556 (2014)
40. Roozbehani, M., Megretski, A., Feron, E.: Optimization of Lyapunov invariants in verification of software systems. IEEE Trans. Autom. Control **58**(3), 696–711 (2013)
41. Roux, P., Voronin, Y.-L., Sankaranarayanan, S.: Validating numerical semidefinite programming solvers for polynomial invariants. In: Rival, X. (ed.) SAS 2016. LNCS, vol. 9837, pp. 424–446. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53413-7_21
42. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Non-linear loop invariant generation using gröbner bases. ACM SIGPLAN Not. **39**(1), 318–329 (2004)
43. Sassi, M.A.B., Girard, A.: Controller synthesis for robust invariance of polynomial dynamical systems using linear programming. Syst. Control Lett. **61**(4), 506–512 (2012)
44. Sassi, M.A.B., Girard, A., Sankaranarayanan, S.: Iterative computation of polyhedral invariants sets for polynomial dynamical systems. In: 2014 IEEE 53rd Annual Conference on Decision and Control (CDC), pp. 6348–6353. IEEE (2014)
45. Schaich, R.M., Cannon, M.: Robust positively invariant sets for state dependent and scaled disturbances. In: 2015 IEEE 54th Annual Conference on Decision and Control (CDC), pp. 7560–7565. IEEE (2015)
46. 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. ACM (2011)
47. Tahir, F., Jaimoukha, I.M.: Robust positively invariant sets for linear systems subject to model-uncertainty and disturbances. IFAC Proc. Vol. **45**(17), 213–217 (2012)
48. Tiwari, A.: Termination of linear programs. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 70–82. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27813-9_6
49. Topcu, U., Packard, A., Seiler, P.: Local stability analysis using simulations and sum-of-squares programming. Automatica **44**(10), 2669–2675 (2008)
50. Topcu, U., Packard, A.K., Seiler, P., Balas, G.J.: Robust region-of-attraction estimation. IEEE Trans. Autom. Control **55**(1), 137–142 (2010)
51. Trodden, P.: A one-step approach to computing a polytopic robust positively invariant set. IEEE Trans. Autom. Control **61**(12), 4100–4105 (2016)

52. Vandenberghe, L., Boyd, S.: Semidefinite programming. SIAM Rev. **38**(1), 49–95 (1996)
53. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 154–170. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_11
54. Xia, B., Yang, L., Zhan, N., Zhang, Z.: Symbolic decision procedure for termination of linear programs. Formal Aspects Comput. **23**(2), 171–190 (2011)
55. Yang, L., Zhou, C., Zhan, N., Xia, B.: Recent advances in program verification through computer algebra. Front. Comput. Sci. China **4**(1), 1–16 (2010)

Developing GUI Applications in a Verified Setting

Stephan Adelsberger[1(✉)], Anton Setzer[2], and Eric Walkingshaw[3]

[1] Department of Information Systems, Vienna University of Economics, Vienna, Austria
stephan.adelsberger@wu.ac.at
[2] Department of Computer Science, Swansea University, Swansea, UK
[3] School of EECS, Oregon State University, Corvallis, USA

Abstract. Although there have been major achievements in verified software, work on verifying graphical user interfaces (GUI) applications is underdeveloped relative to their ubiquity and societal importance. In this paper, we present a library for the development of verified, state-dependent GUI applications in the dependently typed programming language Agda. The library uses Agda's expressive type system to ensure that the GUI, its controller, and the underlying model are all consistent, significantly reducing the scope for GUI-related bugs. We provide a way to specify and prove correctness properties of GUI applications in terms of user interactions and state transitions. Critically, GUI applications and correctness properties are not restricted to finite state machines and may involve the execution of arbitrary interactive programs. Additionally, the library connects to a standard, imperative GUI framework, enabling the development of native GUI applications with expected features, such as concurrency. We present applications of our library to building GUI applications to manage healthcare processes. The correctness properties we consider are the following: (1) That a state can only be reached by passing through a particular intermediate state, for example, that a particular treatment can only be reached after having conducted an X-Ray. (2) That one eventually reaches a particular state, for example, that one eventually decides on a treatment. The specification of such properties is defined in terms of a GUI application simulator, which simulates all possible sequences of interactions carried out by the user.

Keywords: Agda · Interactive theorem proving · GUI verification

1 Introduction

Graphical user interfaces (GUIs) are widely used in real-world software systems. They are also a major source of bugs. For example, a study of the Mozilla project found that the web browser's GUI is the source of 50.1% of reported bugs and responsible for 45.6% of crashes [33]. Unfortunately, testing of GUIs is notoriously difficult [21,23]. The fundamental problem is that "tests must be automated, but GUIs are designed for humans to use" [30]. Automated tests must simulate user interactions, but the range of user interactions is huge, and simulated actions tend to be brittle with respect to minor changes in the GUI such as swapping the placement of two buttons.

A challenge related to GUIs is the widespread use of code generation. For example, a GUI builder enables a GUI to be graphically assembled, then generates code for the GUI which can be integrated into the rest of the application [34]. Code generation can have a negative impact on software evolution and maintenance if the GUI specification and the program logic cannot be evolved together [36]. In the worst case, handwritten customizations must be manually integrated each time the GUI code is re-generated [34]. Moreover, the dependence of handwritten code on the upstream specification is implicit.

Because of their importance and the challenges related to testing them, researchers have studied the formal verification of GUI applications using model checking [22]. However, such approaches verify only an abstracted model of the GUI rather than the software itself.

In this paper, we present a library for developing directly verified GUI applications in Agda [6]. Agda is a dependently typed programming language and interactive theorem prover. Our library supports verifying properties about GUI applications in terms of user interactions. We address the challenge of code generation by observing that such systems are *implicitly* working with *dependent types* (the hand-written parts depend on the types generated by the declarative specification), and so make this relationship *explicit*. This combines the benefits of a declarative specification with the flexibility of post-hoc customization.

Our library builds on our previous work on state-dependent object-based programming in Agda [1], which is briefly summarized in Sect. 2. The library itself is introduced in Sect. 3, beginning with an example of a GUI application written using the library, followed by a description of key aspects of the implementation. In our library, GUIs are declaratively specified as a value of an inductive data type. This *value* is used in the *types* of the controller functions that link the GUI to the underlying business logic, guaranteeing via the type system that the controller code is consistent with the GUI specification. Since the GUI specification is an inductive value, we can write arbitrary functions to modify it, improving the flexibility of GUI specifications.

In Sect. 5.1, we present a case study based on a healthcare process adapted from the literature [24]. Processes in the healthcare domain are mostly data-driven (e.g. patient data), include interactions with doctors (e.g. diagnosis), and are not easily modeled as finite state machines [24,26]. We demonstrate that by using dependent types, we can model such state-dependent systems with an infinite number of states.

On this note, in Sects. 4 and 5, we develop a framework for specifying and proving the correctness of GUI applications. More precisely, in Sect. 4, we define a simulator that simulates sequences of user inputs to a GUI. Using this, we define *reachability* as whether there exists a sequence of user inputs to get from one state of the GUI to another. We then conduct a small case study to prove for a simple GUI application which states can be reached from a given state. In Sect. 5, we develop an example from the healthcare domain with interactive handlers (i.e. the observer pattern where an object handles GUI events) and a data-dependent GUI. We specify an *intermediate-state property* that requires passing through a specific state before reaching some other state, and we prove that the example satisfies a given property. Finally, we specify a *final-state property* that requires the application to eventually reach a given state from a start state. We show that the advanced example fulfills such a property. It turns out that the complexity lies more in the specification of the properties, while proving the properties is relatively straightforward. We discuss related work in Sect. 6 and conclusions in Sect. 7.

To summarize, the contributions of this paper are:

1. A dependently typed library for programming state-dependent GUI applications, which allows an infinite number of states and arbitrary interactive handlers.
2. A technique for specifying properties of GUIs using a simulator which simulates all possible sequences of interactions carried out by the user.
3. A framework for the verification of GUI applications involving: (1) reachability between states, (2) that one state can only be reached by passing through another state, and (3) that one eventually reaches a specific state.

Source Code. All displayed Agda code has been extracted automatically from type checked Agda code. For readability, we have hidden some bureaucratic details and show only the crucial parts of the code. The code is available at [4].

2 Background

Agda and Sized Types. We recommend the short introduction to Agda that we provided in a previous paper [1]; here, we will just repeat the basics. Agda [6] is a theorem prover and dependently typed programming language based on Martin-Löf type theory. Propositions are represented as types, and a value of a type corresponds to a proof of the preposition. Agda features a type checker, a termination checker, and a coverage checker. The termination and coverage checker guarantee that every program in Agda is total. Partiality would make Agda inconsistent as a proof language.

Agda has a hierarchy of types for describing sets of types themselves, for example, the set of all types in the usual sense is called Set in Agda. The type Set_1 includes all of Set plus types that refer to Set itself. For example, in our library, we use Set_1 to define structures that have Set as one of its components (e.g. IOInterface below). Agda has function types, inductive data types, record types, and dependent function types. A dependent function type $(x : A) \rightarrow B$ represents the type of functions mapping an element $x : A$ to an element of B where B may depend on x. The variant form $\{x : A\} \rightarrow B$ of the dependent function type allows us to omit argument x when applying the function; Agda will try to infer it from typing information, but we may still apply it explicitly as $\{x = a\}$ or $\{a\}$ if Agda cannot deduce it automatically.

To represent infinite structures we use Agda's coinductive record types, equipped with size annotations [16]. The size annotations are used to show productivity of corecursive programs [2], which we define using copattern matching [3].

State-Dependent IO. In previous work [1], we gave a detailed introduction to interactive programs, objects, and state-dependent versions of interactive programs and objects in dependent type theory. The theory of objects in dependent type theory is based on the IO monad in dependent type theory, developed by Peter Hancock and the second author of this article [14]. The theoretical basis for the IO monad was developed by Moggi [25] as a paradigm for representing IO in functional programming. The idea of the IO monad is that an interactive program has a set of commands to be executed in the real world. It iteratively issues a command and chooses its continuation depending on the response from the real world. Formally, our interactive programs are coinductive, i.e. infinitely deep, Peterson-Synek trees [28], except that they also have the option to terminate and return a value. This allows for monadic composition of programs, namely sequencing one program with another program, where the latter program depends on the return value of the first program. In the state-dependent version [1], both the set of available commands and the form of responses can depend on a state, and commands may modify the state.

We introduce now an IO language for GUIs. It will include also console commands and calls to external services, such as database queries. An IO language, which we call IO interface, is a record consisting of commands a program can issue, and responses the real world returns in response to these commands.

```
record IOInterface : Set₁ where
  Command  : Set
  Response : (m : Command) → Set
```

The fields of this record type Command and Response are its projections. They can be applied also postfix using the dot notation: if P: IOInterface, then P: .Command : Set. To improve readability we omit in records bureaucratic statements field, coinductive, and open.

3 State-Dependent GUI Applications

Our library separates the structure and appearance of an application's GUIs (the view) from the *handlers* that process the events produced by user interactions (the controller). This separation of concerns is similar to current practice with model-view-controller frameworks [20] and graphical GUI-builder tools [34]. A distinguishing feature of our approach is that handlers are dependently typed with respect to the GUIs they interface with. This means that GUI specifications can be programmatically generated and dynamically modified (e.g. a button may be dynamically added at runtime) without sacrificing the static guarantee of consistency with the handler objects. As a GUI dynamically changes, the interfaces of the corresponding handler objects (which methods exist and their types) dynamically change in response. Such dynamically changing GUIs are not well supported by the GUI-builder model, and the consistency guarantees are not provided by programmatic MVC frameworks.

In the rest of this section, we introduce our library. In Sect. 3.1, we provide an introductory example that illustrates how to build a simple state-dependent GUI application. In Sect. 3.2, we turn to the implementation of the library, introducing the basic command interface provided by the library for creating and interacting with GUIs. In previous work, we have developed a representation of state-dependent objects, which we briefly describe in Sect. 3.3. This representation is the basis for GUI event handlers in our library. We say that our library supports *state-dependent* GUI applications since the GUI can dynamically change based on the state of the model and since the GUI is itself a dynamically changing state of the handler objects. We use state-dependent objects to define generic handlers in Sect. 3.4 whose interfaces are generated from the GUIs they depend on. We also introduce a data type that collects all of the components of a GUI application together. Finally, in Sect. 3.5, we introduce an example of a GUI having infinitely many states, where each state differs in the GUI elements used.

3.1 Introductory Example

Consider a GUI application with two states. In one state, the GUI has a single button labelled “OK”, in the other it also has a button labelled “Cancel”. Pressing the OK button in either state switches to the other state.

Creating a GUI application in our library consists of specifying the GUI, including defining the GUI elements and their properties (e.g. the color of a button), then defining the handler objects for events those elements can produce. For our example, we first specify the GUI for the first state, which consists of a frame and a single button labelled “OK”. Then, we specify the GUI for the second state by adding a button labelled “Cancel” to oneBtnGUI. Both oneBtnGUI and twoBtnGUI are of type Frame.

```
oneBtnGUI = addButton "OK" create-frame
twoBtnGUI = addButton "Cancel" oneBtnGUI
```

Finally, we specify the initial properties of GUIs (e.g., properties for twoBtnGUI), specifically the color of the button labels (black) and the frame layout.

```
propOneBtn : properties oneBtnGUI
propOneBtn = black , oneColumnLayout

propTwoBtn : properties twoBtnGUI
propTwoBtn = black , black , oneColumnLayout
```

Observe that the types of the property specifications are dependent on the corresponding GUI values, ensuring consistency between the two.

Next we define handlers for our application's two GUI states. The handler for oneBtnGUI defines a method that handles the event generated by clicking the OK button. The handler body is an interactive IO program that prints a string indicating the button click, then uses the function changeGUI to set the currently active GUI to twoBtnGUI, updating the properties and handler accordingly.

```
obj1Btn : ∀ {i} → HandlerObject i oneBtnGUI
obj1Btn .method bt = putStrLn "OK! Redefining GUI." »
                     changeGUI twoBtnGUI propTwoBtn obj2Btn
```

This is an example of copattern matching [3]: The type of obj1Btn is a record which has one field method. We define obj1Btn by determining the value for this field, which is in postfix notation written as .method. The result is a function having further argument *bt*, so we apply it to this argument and define the result. Although unused in obj1Btn, *bt* provides access to enclosing GUI elements, such as the GUI's frame.

Note that the type of the handler object HandlerObject is parameterized by the corresponding GUI value, in this case, oneBtnGUI. The type of the handler object ensures that it has methods available to handle each kind of event that the given GUI can generate.

The hidden argument $\{i\}$ is a size parameter required since handler objects are coinductive data types. It is used to check that the definition of the handler is productive, which is the copattern matching dual to termination checking.

The handler for twoBtnGUI defines a method that uses pattern matching on the argument *bt* to determine which button was clicked; this is an example of combined pattern/copattern matching.

```
obj2Btn : ∀ {i} → HandlerObject i twoBtnGUI
obj2Btn .method (firstBtn bt) = putStrLn "OK! Redefining GUI." »
                                changeGUI oneBtnGUI propOneBtn obj1Btn
obj2Btn .method (secondBtn bt) = putStrLn "Cancel!" » keepGUI obj2Btn
```

If the click originated from the first button (OK), the active GUI is changed back to oneBtnGUI. Otherwise, if it originated from the second button (Cancel), then twoBtnGUI is retained using the library function keepGUI.

Finally, we can compile our GUI application into a NativeIO program in Agda. NativeIO is a type that represents IO programs in Haskell. This is done by calling compileProg with the arguments twoBtnGUI, propTwoBtn, and obj2Btn. The compiler of Agda translates the resulting NativeIO program via special Haskell FFI commands present in Agda into a GUI Haskell program which makes use of the wxHaskell [37] GUI toolkit.

3.2 GUI Interface

The type GuiInterface defines commands for interacting with the console, the GUI (e.g., changing a label), and communicating with a database, and the responses to those commands. Here, we only list the commands used later in this paper (console commands), see the repository [4] for the full list:

```
data GuiCommand : Set where
  putStrLn : String → GuiCommand
  getLine   : GuiCommand

GuiResponse : GuiCommand → Set
GuiResponse getLine = String
GuiResponse _        = Unit

GuiInterface : IOInterface
GuiInterface .Command = GuiCommand
GuiInterface .Response  = GuiResponse
```

3.3 State-Dependent Interfaces

The handling of GUI commands will be implemented with state-dependent objects where the interface may change according to the state of the object. An object is similar to an interactive program except that it is a server-side program. An object can receive commands, and depending on the responses to those commands it returns a result and changes its internal state. With Agda being a purely functional language, we model state changes by returning an updated object together with the regular return value as a pair. In dependent type theory we also have state-dependent objects, where the methods available depend on the state of the object. Depending on the response for the method call, the object switches to a new state. So an interface for a state-dependent interface $\mathsf{Interface}^{s}$ (^{s} indicating that it is state-dependent) consists of a set of states, a set of methods depending on the state, a set of responses depending on states and methods, and a next state function, which depends on states, methods and responses:

```
record Interface^s : Set_1 where
  State    : Set
  Method : State → Set
  Result   : (s : State) → (m : Method s) → Set
  next      : (s : State) → (m : Method s) → (Result s m) → State
```

We note here that a method in this framework is what corresponds to the union of all the methods together with all their arguments in normal object-oriented languages. The reason why we can bundle them together is because the result type can depend on the method, therefore there is no need for separate methods with separate result types. An Object for this interface is a program that accepts calls to objects methods (method). In response to an object method, it returns a result and an updated Object. Since this interaction might go on forever, the definition of an Object is coinductive.

3.4 Implementation of Generic GUIs

In the previous subsection, we saw that the type of a handler object depends on the value of the GUI it supports. To help understand how handler objects work, we start by taking a closer look at the type of generic handler objects.

```
HandlerObject : ∀ i → Frame → Set
HandlerObject i g = IOObjectˢ GuiInterface handlerInterface i g
```

This library function defines the type of a handler object, given a size index i and a frame that represents the GUI the handler processes events for. As described in Sect. 3.1, i is used to ensure that the handler is productive, a well-formedness property of co-inductive definitions.

The type of a handler is a state-dependent object that supports GuiInterface commands (see Sect. 3.2). The state of the object is the GUI specification; it is parameterized by a Frame, which determines the interface of the object as defined below.

```
handlerInterface : Interfaceˢ
handlerInterface .State        = Frame
handlerInterface .Method f     = methodsG f
handlerInterface .Result f m   = returnType f
handlerInterface .next f m r   = nextStateFrame f r
```

We need a type for the methods, which depend on the frame. As mentioned before all individual methods of an object are bundled together into one single one. For each individual GUI component such as a button, and each event corresponding to this component, we require one method for handling it. The method for a component is the sum of all the methods for its events, and the methodsG function creates the sum of all the methods of each component of the frame. When an individual event is called, it obtains as arguments the parameters of this call. Since the parameters are part of the method, an event method is the product of the parameters of this method call together with an element representing all the components of the frame. It is the task of the user to implement these methods, when he creates a handler object for a frame definition.

Since a handler object's interface (i.e. what methods it provides to handle GUI events) is determined by its state, the interface dynamically updates with corresponding changes to the GUI specification.

An event handler method is an interactive program that has three possible return options:

```
data returnType (f : Frame) : Set where
  noChange          : returnType f
  changedAttributes : properties f → returnType f
  changedGUI        : (fNew : Frame) → (properties fNew) → returnType f
```

In the first case, the GUI remains unchanged. In the second case, we simply return the changed properties. In the third case we transform the given Frame into a newly created GUI.

The function nextStateFrame carries out the calculation of the new successor state after a method is finished. The state is updated only in case of the return option changedGUI.

```
nextStateFrame : (f : Frame)(r : returnType f) → Frame
nextStateFrame f noChange                = f
nextStateFrame f (changedAttributes x) = f
nextStateFrame f (changedGUI fNew x)   = fNew
```

3.5 A GUI with an Unbounded Number of States

In this subsection, we present an introductory example that both illustrates the use of the GUI data type and demonstrates that we can develop GUIs with infinitely many states where each state differs in the GUI elements used. The example is a GUI application with n buttons, where clicking any button expands the GUI into one with $n + 1$ buttons.[1] The helper function nFrame constructs a GUI with n buttons, while nProp defines its corresponding properties (a black label for each button organized in a one-column layout). The nGUI function constructs a GUI application with n buttons combining the GUI, its properties, and a handler that for any button press replaces the GUI application with a new one containing $n + 1$ buttons.

```
nFrame : (n : ℕ) → Frame
nFrame 0       = create-frame
nFrame (suc n) = addButton (show n) (nFrame n)

nProp : (n : ℕ) → properties (nFrame n)
nProp 0       = oneColumnLayout
nProp (suc n) = (black , nProp n)

nGUI : ∀{i} → (n : ℕ) → GUI {i}
nGUI n .defFrame         = nFrame n
nGUI n .property         = nProp n
nGUI n .obj .method m = changeGUI (nGUI (suc n))
```

The type GUI represents a GUI application. It is a record with three fields: .defFrame defines the GUI, .property specifies its properties, and .obj contains the GUI's handler object, whose field .method is invoked when a button is clicked. The argument m to the handler method indicates which button was clicked, but in this case, we ignore it since clicking any button updates the GUI to add one more button. Note that the GUI application defined above is dynamically expanding, which is difficult to design using standard GUI builders since they allow constructing only finitely many GUIs for a particular application.

[1] The library contains a more interesting example [4] where clicking button b_i extends the GUI with i additional buttons.

4 Proof of Correctness Properties of GUIs

We reason about GUI applications by reasoning about the GUI's states. The state of a GUI is given by a frame, its properties, and its handler object. When an event is triggered, an IO program is executed, and the return value determines which state to transition to. Thus, we can reason about the transition graph for a GUI application by reasoning about the exit points of the handler. However, a complication is that IO programs are coinductive, meaning they may have an unbounded number of interactions and never terminate. Ideally, IO programs for event handlers would be inductive since we typically want event handlers to always terminate so that the GUI is responsive. However, this is much more difficult to integrate within a general GUI framework since GUI applications are naturally coinductive.

4.1 A Simulator for GUI Applications

To cope with coinductive IO programs in event handlers, we do not reason about GUI states directly, but instead introduce an intermediate model of a GUI application, where the IO programs in handlers are unrolled into potentially infinitely many states. This model is itself coinductive and we cannot reason about it directly since an infinite sequence of IO commands will induce an infinite number of states. Therefore, we instead reason about *finite simulations* of the GUI model.

To define the model, we first introduce a data type to indicate whether an event handler has been invoked or not. The notStarted constructor indicates that the handler has not yet been invoked, while started indicates the handler has been invoked. The constructor started has as an additional argument *pr* corresponding to the IO program still to be executed.

```
data MethodStarted (f : Frame) (prop : properties f)
                   (obj : HandlerObject ∞ f) : Set where
  notStarted : MethodStarted f prop obj
  started    : (m : methodsG f) (pr : IO' GuiInterface ∞ StateAndGuiObj)
             → MethodStarted f prop obj
```

The handler is parameterized by the size value ∞. Sizes are ordinals that limit the number of times a coinductive definition can be unfolded. There is an additional size ∞ for coinductive definitions that can be unfolded arbitrarily many times. A more detailed explanation of sizes and ∞ can be found in Sect. 3 of [17].

Now, a state in the GUI model can be represented by the GUI, its properties, the handler, and the invocation state.

```
data ModelGuiState : Set where
  state : (f : Frame) (prop : properties f) (obj : HandlerObject ∞ f)
          (m : MethodStarted f prop obj) → ModelGuiState
```

Using this model, we can simulate the execution of GUI applications. To do this, we define a simulator for state-dependent IO programs. Depending on the state of the GUI model, the simulator must trigger GUI events or provide responses to IO commands, then move to the next state in the model. The following function defines the available actions at each state in the model.

```
modelGuiCommand : (s : ModelGuiState) → Set
modelGuiCommand (state g prop obj notStarted)              = methodsG g
modelGuiCommand (state g prop obj (started m (exec' c f))) = GuiResponse c
modelGuiCommand (state g prop obj (started m (return' a))) = ⊤
```

If the model is in a notStarted state, the event simulator can trigger an event drawn from the methods supported by the GUI interface. If the model is in a started state, then there are two sub-cases: If the IO program has not finished, the program has the form (exec' c f). This means that the next real-world command to be executed is c and once the world has provided an answer r to it, the interactive program continues as (f r). (Previously we used do instead of exec, but do has now become a keyword in Agda.) In this case the GUI is waiting on a response to the IO command c, which the event simulator must provide. The second subcase is, if the remaining IO program has already returned. Then the simulator can take the trivial action (⊤) to return to the notStarted state.

Using this definition, we can define a transition function for the simulator with the following type:

```
modelGuiNext : (s : ModelGuiState) (c : modelGuiCommand s) → ModelGuiState
```

That is, given a model state and an action of the appropriate type, we can transition to the next model state.

To simplify proofs over the model, the transition function makes a few optimizations. First, in the case where the new state corresponds to a completed IO program, we can skip to the next notStarted state directly rather than requiring this unit step be made explicitly. Second, we reduce sequences (shorter than a given finite length) of consecutive trivial IO actions, such as print commands, into single transition steps.

We can define a state-dependent IO interface (an element of $\mathsf{IOInterface}^{\mathsf{s}}$) for the simulator (see Sect. 2), which incorporates the previous definitions in a straightforward way.

```
modelGuiInterface : IOInterfaceˢ
modelGuiInterface .State           = ModelGuiState
modelGuiInterface .Command         = modelGuiCommand
modelGuiInterface .Response s m    = ⊤
modelGuiInterface .next     s m r  = modelGuiNext s m
```

Using this, we define a relation between states s and s' of the GUI, which states that s' is reachable from s if running the simulator from s can produce s'.

$$\begin{aligned}&\_\text{-gui->}\_ : (s\ s' : \mathsf{ModelGuiState}\ ) \to \mathsf{Set}\\ &s \text{ -gui-> } s' = \mathsf{IO}^{\mathsf{s}}\mathsf{ind}_{p0}\ \mathsf{modelGuiInterface}\ s\ s'\end{aligned}$$

Here, s -gui-> s' expresses that we can get from s to s' in a finite number of steps. It is therefore defined inductively which allows to reason about it inductively. Finally, we introduce a one-step relation that states that s is reachable from s' by executing one step of the simulator.

$$\begin{aligned}&\mathsf{data}\ \_\text{-gui->}^1\_\ (s : \mathsf{ModelGuiState}\ ) : (s' : \mathsf{ModelGuiState}) \to \mathsf{Set}\ \mathsf{where}\\ &\quad \mathsf{step} : (c : \mathsf{modelGuiCommand}\ s) \to s \text{ -gui->}^1\ \mathsf{modelGuiNext}\ s\ c\end{aligned}$$

The correctness proofs using the simulator are included in the code repository [4].

5 State Transition Properties

In this section, we demonstrate the definition and proof of properties relating to state transitions using an example from the healthcare domain. In Sect. 5.1, we consider the property that any path from one state to another in a GUI application must pass through a given intermediate state. In Sect. 5.2, we consider the property that all paths through a GUI application end up in the same final state. In both cases, the main challenge is to express the property to be proved, while the proof is relatively straightforward.

Such properties are well covered by existing approaches based on model checking [22]. However, the advantage of our approach is that we prove such properties for the implementation of the GUI application directly, rather than for an abstracted model of it.

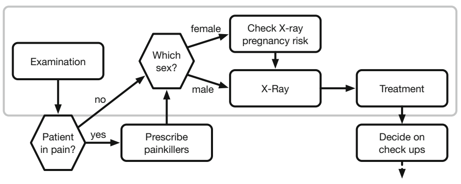


Fig. 1. Process model of a fracture treatment process.

5.1 Intermediate-State Properties

Consider the healthcare process illustrated in Fig. 1, which is adapted from [24]. The relevant part of the process is highlighted. Specifically, it consists of four states corresponding to steps in the process: an initial examination, performing an X-ray, assigning treatment, and a risk check for pregnancy in which the patient is asked about a potential pregnancy. The last state is only performed for female patients. In this subsection, we will build a GUI application abstracting this part of the process and prove that all paths from the initial examination to treatment pass through the intermediate X-ray state. We define a generic mechanism for expressing such intermediate-state properties, which illustrates how other properties on GUI applications can be defined.

Below is the straightforward specification of the GUI for the initial examination state. The GUIs for other states are defined similarly.

```
frmExam : Frame
frmExam = addBtn "Examination" create-frame
```

The controller for the initial examination state is shown below. After pressing the button, the system interactively asks whether the patient is female or male. If the answer is female, the controller invokes changeGUI to change the application to the pregnancy-test state. (A version which includes a check of the correctness of the user input is discussed in [5].) Otherwise, the controller changes to the X-ray state. This is an example of a data-dependent GUI with interaction.

```
hdlExam : ∀ i → HandlerObject i frmExam
hdlExam i .method {j} (btn , frm) =
  exec (putStrLn "Female or Male?") λ _ →
  exec getLine λ s →
  hdlExamProgEnd i (string2Sex s)

hdlExamProgEnd : (i : Size)(g : Sex) → HandlerIOType i frmExam
hdlExamProgEnd i female = changeGUI frmPreg propOneBtn (hdlPreg i)
hdlExamProgEnd i male   = changeGUI frmXRay propOneBtn (hdlXRay i)
```

The GUI and controller for the X-ray state is not shown, but it provides a single button that when pressed transitions to the treatment state.

To reason about our GUI application, we define the corresponding coinductive model as described in Sect. 4.1. We first model the initial examination state as stateExam along with two intermediate states corresponding to interactions with the user: stateExam_1 is the state reached after querying the sex of the patient, while stateExam_2 is the state reached after the user provides a response.

```
stateExam : ModelGuiState
stateExam = state frmExam propOneBtn (hdlExam ∞) notStarted

stateExam₁ : (c : methodsG frmExam) → ModelGuiState
stateExam₁ = modelGuiNext stateExam

stateExam₂ : (c : methodsG frmExam) (str : String) → ModelGuiState
stateExam₂ c str = modelGuiNext (stateExam₁ c) str
```

Similarly, we model the perform-X-ray state as stateXRay and the assign-treatment state as stateTreatm.

We expect that our GUI application implements the healthcare process in Fig. 1. As mentioned, we want to ensure that we never assign a treatment without first performing an X-ray. To support stating such a property, we define a predicate (*path* goesThru*s*′), which expresses that a *path* from states s to s' passes through a state t. The path from s to s' is a member of the type -gui->. The predicate is defined inductively, where the constructors exec$_i$ and return$_i$ are the inductive forms of the usual exec and return constructors for coinductive programs.

```
_goesThru_ : {s s' : ModelGuiState}(q : s -gui-> s')(t : ModelGuiState) → Set
_goesThru_ {s} (execᵢ c f) t = s ≡ t ⊎ f _ goesThru t
_goesThru_ {s} (returnᵢ a) t = s ≡ t
```

In the exec$_i$ case, the current state is either equal to t or all subsequent states must pass through the t. In the return$_i$ case, the path is the trivial path, and therefore the current state must be equal to t.

Now we can show that any path through the GUI application from the initial-examination state to the treatment state passes through the X-ray state. We need to prove this property not only for the initial state but also for all intermediate states. The proof for the initial state is shown below by matching on exec$_i$ and showing that all subsequent states have this property. We match the case return$_i$ with the empty case distinction (indicated by ()), which is always false.

```
examGoesThruXRay : (p : stateExam -gui-> stateTreatm) → p goesThru stateXRay
examGoesThruXRay (execᵢ c f) = inj₂ (exam₁GoesThruXRay c (f _))
examGoesThruXRay (returnᵢ () )
```

Cases for most of the other states are similarly straightforward. However, we show in detail the interesting case of stateExam$_2$ in which we need to make a case distinction on the value of (string2Sex*str*). For this, we use the magic with construct, which allows extending a pattern by matching on an additional expression. The symbol ···| expresses that the previous pattern is repeated, extended by a pattern matching on the with-expression.

```
exam₂GoesThruXRay : (c : methodsG frmExam) (str : String)
  (path : stateExam₂ c str -gui-> stateTreatm) → path goesThru stateXRay
exam₂GoesThruXRay c str path with (string2Sex str)
... | male   = XRayGoesThruXRay path
... | female = checkPregGoesThruXRay path
```

The main difficulty in this proof is to recognize the need for the intermediate states stateExam$_1$ and stateExam$_2$. Once these intermediate states are made explicit and the property is defined, the proof itself is straightforward.

5.2 Final-State Properties

Another property we might want to prove about our healthcare process application is that all paths eventually lead to a treatment. To support the definition and proof of such properties, we define the following inductive data type, which states that all paths from an initial state *start* eventually reach a state *final*.

```
data _-eventually->_ : (start final : ModelGuiState) → Set where
  hasReached : {s : ModelGuiState} → s -eventually-> s
  later : {start final : ModelGuiState} (fornext : (m : modelGuiCommand start)
           → modelGuiNext start m -eventually-> final) → start -eventually-> final
```

The constructors for this type state that the property holds if either the current state is the state to be reached (hasReached) or all subsequent states have this property (later). Since the data type is inductive, this expresses that eventually the final state is reached.

We now show that in the GUI application defined in Sect. 5.1, the treatment state is always reached. Again we prove this property for all states between the initial state and the treatment state. Once again, the interesting case is $\mathsf{stateExam}_2$, where we need to make a case distinction on Sex as before.

```
exam2EventuallyTreatm : (c : methodsG frmExam) (str : String)
   → (stateExam2 c str) -eventually-> stateTreatm
exam2EventuallyTreatm c str with (string2Sex str)
... |   female = checkPregEventuallyTreatm
... |   male   = xRayEventuallyTreatm
```

Here we see the benefits of defining GUIs in a generic way—proving properties about them is straightforward since one can follow the states of the GUI as given by our data type.

6 Related Work

In our previous article [1], we introduced an Agda library for object-based programs. We demonstrated the development of basic, static GUIs. In this paper, we have extended this work by adding a declarative specification of GUIs and the GUI-creation process is now automatic. We also demonstrate the verification of GUI applications in Sects. 4–5.

We have developed an alternative version of this library to use a simpler, custom-built backend [5], rather than the wxHaskell backend used in this paper. The newer backend supports a much simpler version of the GUI interface types described in Sect. 3.2 and the handler objects described in Sect. 3.4 (in fact, much of the complexity of these types is not exposed in this paper for presentation reasons; for full details, see the library [4]), which simplifies proofs and improves the scalability of our approach. However, the more complex approach described in this paper is more generic and has the advantage of being built on an existing and widely used GUI toolkit (wxWidgets). In particular, this version supports GUIs with nested frames, separates properties from the GUIs they apply to, allows modifying properties without redrawing the entire GUI, and supports adding new components from wxWidgets relatively easily. Additionally, the wxHaskell backend supports concurrency and integrates better with the host operating system than our custom backend. The downside is that wxHaskell is an inherently imperative and concurrent toolkit, which makes interfacing with Agda non-trival and leads to the increased complexity of this version (for the technical details, again see the library [4]). In addition, in [5], we add a representation of business processes in Agda and automatically compile such processes into executable GUIs. Taking advantage of the simpler design, in [5] we also implement a larger, realistic case study but proved only reachability statements, whereas in this paper we perform a simpler case study but also cover intermediate state properties.

Functional Reactive Programming (FRP) is another approach for writing GUIs in functional languages [35], where a behaviour is a function from time to values, and an event is a value with occurrence time. In connection with dependent types, FRP has been studied from the foundational perspective [31] and for verified programming [18].

Formlets [12] are abstract user interface that define static, HTML based forms that combine several inputs into one output. However, they do not support sequencing multiple intermediate requests.

Formalizations in Isabelle of end-user-facing applications have been studied for distributed applications [8] and conference management systems [19]. However, only the respective core of the server is verified.

Process Models in the Healthcare Domain were specified using Declare [24], also with extensions to model patient data [26]. In the current paper, in contrast to [11,24,26] and other approaches, we apply formal verification using a theorem prover (Agda) and provide machine-checked proofs. We found only two papers using formal specifications: Debois [13] proves in Isabelle a general result of orthogonality of events. Montali et al. [27] developed models via choreographies. But the latter is limited to finite systems and doesn't deal with interactive events.

Furthermore, healthcare processes are safety critical and testing GUI applications is a major challenge [21]. Sinnig et al. [32] argued that it is important to formalize business processes together with the design information/requirements of the user interface (e.g. GUI) in a single framework. This is directly supported by our library and a benefit of our use of dependently typed GUI programming.

Verification of user interfaces for medical devices. An approach to the verification of user interfaces based on a combination of theorem proving and model checking is presented in [15]. In particular, [15] focuses on the problem of how to demonstrate that a software design is compliant with safety requirements (e.g., certain FDA guidelines). Their solution is elegant in their combined use of model checking, theorem proving, and simulation within one framework. A difference to our work is that [15] verifies models of devices while we verify the software (e.g., handlers of GUI events) directly. Furthermore, we allow the verification of GUI software with an infinite number of states which is not the focus of [15].

Idris and Algebraic Effects. Bauer and Pretnar [7] introduced algebraic effects. Brady [9] adapted this approach to represent interactive programs in Idris [10]. In [1], Sect. 11 we gave a detailed comparison of the IO monad in Agda and algebraic effects in Idris and a translation between the two approaches. Regarding GUIs, we only found a forum post [29], which shows that GUI programming should be possible using the FFI interface of Idris but has yet to be performed.

7 Conclusion and Future Work

Verification of GUI-based programs is important because they are widely used, difficult to test, and many programs are safety critical. We demonstrate a new approach to generically representing and verifying GUI applications in Agda. Our approach makes essential use of dependent types to ensure consistency between the declarative GUI specification and the rest of the system. We demonstrated a generic mechanism for expressing intermediate-state properties. For example, we proved that between the initial examination and the treatment, the GUI application must pass through the intermediate X-ray state. We also considered the property that all paths through a GUI application arrive at a particular final state. Finally, we presented the generation of working GUI applications, including GUI programs with an unbounded number of states.

A limitation of our current approach is that, although the underlying GUI framework supports concurrency, we do not have a way to represent or reason about this explicitly in our library. Concurrency is important for defining GUI applications that remain responsive while a long-running event handler executes. We are working on an extension to our library that allows introducing and reasoning about concurrency, such as defining multiple threads and proving that they are fairly executed.

Acknowledgments. The first and second author were supported by the project CA COST Action CA15123 European research network on types for programming and verification (EUTYPES). The second author was additionally supported by CORCON (Correctness by Construction, FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638), COMPUTAL (Computable Analysis, FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962), CID (Computing with Infinite Data, Marie Curie RISE project, H2020-MSCA-RISE-2016-731143). The third author was supported by AFRL Contract FA8750-16-C-0044 under the DARPA BRASS program.

References

1. Abel, A., Adelsberger, S., Setzer, A.: Interactive programming in Agda - objects and graphical user interfaces. J. Funct. Program. **27**, 38 (2017). https://doi.org/10.1017/S0956796816000319
2. Abel, A., Pientka, B.: Well-founded recursion with copatterns and sized types. JFP **26**, e2 (2016). iCFP 2013 special issue
3. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: programming infinite structures by observations. In: POPL 2013, pp. 27–38. ACM, New York (2013)
4. Adelsberger, S., Setzer, A., Walkingshaw, E.: Deveoping GUI applications in a verified setting (2017). https://github.com/stephanpaper/SETTA18, git respository

5. Adelsberger, S., Setzer, A., Walkingshaw, E.: Declarative GUIs: simple, consistent, and verified. In: International Conference on Principles and Practice of Declarative Programming (PPDP). ACM (2018)
6. Agda Community: Agda Wiki (2017). http://wiki.portal.chalmers.se/agda
7. Bauer, A., Pretnar, M.: Programming with algebraic effects and handlers (2012). http://arxiv.org/abs/1203.1539, arXiv
8. Bauereiß, T., Gritti, A.P., Popescu, A., Raimondi, F.: CoSMeDis: a distributed social media platform with formally verified confidentiality guarantees. In: 2017 Symposium on Security and Privacy, pp. 729–748. IEEE (2017)
9. Brady, E.: Resource-dependent algebraic effects. In: Hage, J., McCarthy, J. (eds.) TFP 2014. LNCS, vol. 8843, pp. 18–33. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14675-1_2
10. Brady, E.: Type-Driven Development with Idris, 1st edn. Manning Publications, Greenwich (2017)
11. Chiao, C.M., Künzle, V., Reichert, M.: Towards object-aware process support in healthcare information systems. In: eTELEMED 2012. IARIA, Delaware (2012)
12. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: The essence of form abstraction. In: Ramalingam, G. (ed.) APLAS 2008. LNCS, vol. 5356, pp. 205–220. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89330-1_15
13. Debois, S., Hildebrandt, T., Slaats, T.: Concurrency and asynchrony in declarative workflows. In: Motahari-Nezhad, H.R., Recker, J., Weidlich, M. (eds.) BPM 2015. LNCS, vol. 9253, pp. 72–89. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23063-4_5
14. Hancock, P., Setzer, A.: Interactive programs in dependent type theory. In: Clote, P.G., Schwichtenberg, H. (eds.) CSL 2000. LNCS, vol. 1862, pp. 317–331. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44622-2_21
15. Harrison, M.D., Masci, P., Campos, J.C., Curzon, P.: Verification of user interface software: the example of use-related safety requirements and programmable medical devices. IEEE Trans. Hum.-Mach. Syst. **47**(6), 834–846 (2017)
16. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: POPL 1996, pp. 410–423. ACM, New York (1996)
17. Igried, B., Setzer, A.: Defining trace semantics for CSP-Agda, 30 January 2018. http://www.cs.swan.ac.uk/~csetzer/articles/types2016PostProceedings/igriedSetzerTypes2016Postproceedings.pdf. Accepted for Publication in Postproceedings TYPES 2016, 23 p.
18. Jeffrey, A.: LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In: PLPV 2012. ACM, New York (2012)
19. Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 167–183. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_11
20. Krasner, G.E., Pope, S.T.: A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. JOOP **1**(3), 26–49 (1988)
21. Memon, A.M.: GUI testing: pitfalls and process. Computer **35**(8), 87–88 (2002)
22. Memon, A.M.: An event-flow model of GUI-based applications for testing. Softw. Test. Verif. Reliab. **17**(3), 137–157 (2007)
23. Memon, A.M., Xie, Q.: Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. IEEE Trans. Softw. Eng. **31**(10), 884–896 (2005)

24. Mertens, S., Gailly, F., Poels, G.: Enhancing declarative process models with DMN decision logic. In: Gaaloul, K., Schmidt, R., Nurcan, S., Guerreiro, S., Ma, Q. (eds.) CAISE 2015. LNBIP, vol. 214, pp. 151–165. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19237-6_10
25. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991)
26. Montali, M., Chesani, F., Mello, P., Maggi, F.M.: Towards data-aware constraints in Declare. In: SAC 2013, pp. 1391–1396. ACM (2013)
27. Montali, M., Pesic, M., van der Aalst, W.M.P., Chesani, F., Mello, P., Storari, S.: Declarative specification and verification of service choreographies. ACM Trans. Web **4**(1), 3:1–3:62 (2010)
28. Petersson, K., Synek, D.: A set constructor for inductive sets in Martin-Löf's type theory. In: Pitt, D.H., Rydeheard, D.E., Dybjer, P., Pitts, A.M., Poigné, A. (eds.) Category Theory and Computer Science. LNCS, vol. 389, pp. 128–140. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0018349
29. Pinson, K.: GUI programming in Idris? (2015). https://groups.google.com/forum/#!topic/idris-lang/R_7oixHofUo, google groups posting
30. Ruiz, A., Price, Y.W.: Test-driven GUI development with TestNG and Abbot. IEEE Softw. **24**(3), 51–57 (2007)
31. Sculthorpe, N., Nilsson, H.: Safe functional reactive programming through dependent types. In: ICFP 2009, pp. 23–34. ACM (2009)
32. Sinnig, D., Khendek, F., Chalin, P.: Partial order semantics for use case and task models. Formal Asp. Comput. **23**(3), 307–332 (2011)
33. Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y., Zhai, C.: Bug characteristics in open source software. Empir. Softw. Eng. **19**(6), 1665–1705 (2014)
34. Valaer, L.A., Babb, R.G.: Choosing a user interface development tool. IEEE Softw. **14**(4), 29–39 (1997)
35. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: PLDI 2000, pp. 242–252. ACM, New York (2000)
36. Whittle, J., Hutchinson, J., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Softw. **31**(3), 79–85 (2014)
37. wiki: WxHaskell. https://wiki.haskell.org/WxHaskell. Accessed 9 Feb 2017

Interleaving-Tree Based Fine-Grained Linearizability Fault Localization

Yang Chen[1,2], Zhenya Zhang[3], Peng Wu[1,2](✉), and Yu Zhang[1,2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
wp@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] National Institute of Informatics, Tokyo, Japan

Abstract. Linearizability is an important correctness criterion for concurrent objects. Existing work mainly focuses on linearizability verification of coarse-grained traces with operation invocations and responses only. However, when linearizability is violated, such coarse-grained traces do not provide sufficient information for reasoning about the underlying concurrent program faults. In this paper, we propose a notion of *critical data race sequence* (*CDRS*), based on our fine-grained trace model, to characterize concurrent program faults that cause violation of linearizability. We then develop a labeled tree model of interleaved program executions and show how to identify *CDRS*es and localize concurrent program faults automatically with a specific node-labeling mechanism. We also implemented a prototype tool, FGVT, for real-world Java concurrent programs. Experiments show that our localization technique is effective, i.e., all the *CDRS*es reported by FGVT indeed reveal the root causes of linearizability faults.

Keywords: Linearizability · Bug localization · Concurrency Testing

1 Introduction

Localization of concurrency faults has been a hot topic for a long time. Multiple trials on a concurrent program with the same inputs may result in nondeterministic outputs. Hence, it is non-trivial to decide whether a concurrent program is buggy. Moreover, even if a concurrent program is known to contain a bug, it is difficult to reproduce the bug or to determine its root cause.

Efforts have been devoted to addressing the challenge of the localization of concurrency faults. The very basic way is to exhaustively explore the *thread schedule* space to replay and analyze the buggy executions. A thread schedule is usually described as a sequence of thread identifiers that reflects the order

Z. Zhang—The work was partially done when the author was a student at the Institute of Software, Chinese Academy of Sciences.

of thread executions and context switches. In [5,7,26], the thread schedule in a buggy execution is recorded and then replayed to reproduce the same bug. An execution of a concurrent program can be represented as a *fine-grained trace*, which is defined as a sequence of memory access instructions with respect to a specific thread schedule. In [14], fine-grained traces and correctness criteria are encoded as logical formulas to diagnose and repair concurrency bugs through model checking. Generally speaking, such fine-grained analysis suffers from the well-known state space explosion problem. Acceleration techniques have been presented to address this problem with, e.g., heuristic rules [2] or iterative context bounding [20]. However, most of these works aim at general concurrency faults, without cares about their nature or root causes.

In this paper, we focus on linearizability faults. Linearizability [12] is a widely accepted correctness criterion for concurrent data structures or concurrent objects. Intuitively, it means that every operation on a shared object appears to take effect instantaneously at some point, known as a *linearization point*, between the invocation and response of the operation, and the behavior exposed by the sequence of operations serialized at their linearization points must conform to the sequential specification of the shared object.

More attentions have been paid on linearizability verification recently [3,4, 13,17,18]. In these works, an execution of a concurrent program is represented as a *coarse-grained trace*, which is defined as a partial order set of object methods. The basic approach of linearizability checking is to examine whether all the possible topologically sorted sequential traces satisfy the correctness criterion of the shared object. This approach also suffers from the state space explosion problem. Acceleration strategies have been proposed in these works to address this problem, too. However, since the coarse-grained trace model concerns only method invocations and responses, these techniques cannot determine the root causes of linearizability faults.

It is worth mentioning that data races and linearizability faults are different but related concepts. The occurrence of linearizability faults is due to the existence of data races. But not all the data races are critical to the linearizability faults. In this paper, we propose a notion called *Critical Data Race Sequence* (*CDRS*) based on the fine-grained trace model. Intuitively, a CDRS contains a sequence of data races that can decisively cause a linearizability fault in a concurrent program. Thus, the existence of a CDRS implies that the concurrent program can potentially produce a non-linearizable trace. In order to identify a CDRS, we model all the possible fine-grained traces of a concurrent execution as an *interleaving tree*, where each node corresponds to a data race, and each path from the root to a leaf node corresponds to a fine-grained trace. We label each node with a pre-defined symbol, depending on the linearizability of all the paths passing through the node. Then, the existence of a CDRS can be determined based on certain pattern of the node sequences in the labeled interleaving tree.

In order to overcome the state space explosion problem, we divide the localization process into two levels: the coarse-grained level and the fine-grained level. On the coarse-grained level, a *minimum* test case is to be synthesized that

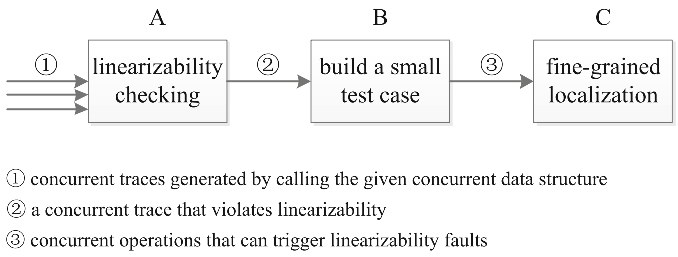

A
B
C
①
linearizability checking
②
build a small test case
③
fine-grained localization
① concurrent traces generated by calling the given concurrent data structure
② a concurrent trace that violates linearizability
③ concurrent operations that can trigger linearizability faults

Fig. 1. Labels of nodes

Table 1. Abbreviations

CDRS	HLDR	CAS	FGVT
Critical Dara Race Sequence	High Level Data Race	Compare And Swap	Fine-Grained VeriTrace

contains a sufficiently small number of operations to trigger a linearizability fault [29]. With such a small test case, the number of memory access instructions examined at the fine-grained level is greatly reduced. Together with the linearizability checking technique [18] and the coarse-grained linearizability fault localization technique [29], the overall localization process is shown in Fig. 1, where stage C is concerned by this paper.

For the brevity of presentation, Table 1 lists the abbreviations used throughout the paper.

Contributions. The main contributions of this paper are as follows:

– We extend the traditional coarse-grained trace model to a *fine-grained trace* model by including memory access events. We also extend the notion of linearizability onto fine-grained traces.
– We propose the notion of *Critical Data Race Sequence* (CDRS), which plays a key role in characterizing the data races that can decisively result in linearizability faults.
– We develop a labeled *interleaving tree* model that contains all the possible fine-grained traces of a concurrent execution. Each node is marked automatically in a way that can reflect the existence of a CDRS through certain pattern.
– We implement a prototype tool, *FGVT*, for real-world Java concurrent programs. Experiments show that FGVT is rather effective in that all the CDRSes reported by FGVT indeed reveal the root causes of linearizability faults.

Related Work. Automated linearizability checking algorithms have been presented in [4,28], but exhibit a performance bottleneck. Based on [28], optimized algorithms have been proposed in [13,18] through partial order reduction and compositional reasoning, respectively. Model checking has been adopted in [3,8] for linearizability checking with simplified first-order formulas, which can help improve efficiency. Fine-grained traces have been introduced in [17] to accelerate linearizability checking. These works lay a firm foundation for the localization of linearizability faults.

Active testing approaches have been proposed for concurrency bug localization based on bug patterns. The characteristics of bug patterns have been discussed in [9,19] in details. Memory access patterns have been proposed in [16,23,24] for ranking bug locations. A fault comprehension technique has further been presented in [22] for bug patterns. Definition-use invariants have been presented in [25] for detecting concurrency bugs through pruning and ranking methods. A constraint-based symbolic analysis method has been proposed in [14] to diagnose concurrency bugs.

Some other concurrency bug localization techniques are based on the bug localization techniques for sequential programs. In [10], concurrent predicates are derived by an assertion mechanism to determine whether a data race causes a concurrency bug. Concurrent breakpoints, an adaption of the breakpoint mechanism, have been proposed in [21] for concurrent program debugging.

This paper aims at linearizability faults in fine-grained traces. It present a novel localization approach based on interleaving trees, which we propose for characterizing fine-grained traces in an effective manner. Hence, it opens a possibility to exploit the existing tree-based search techniques, e.g., partial order reduction, for efficient localization of the linearizability faults.

Organization. The rest of the paper is organized as follows. Section 2 presents an example to illustrate our motivation. Section 3 introduces our fine-grained trace model. Section 4 presents the key notion of CDRS based on our fine-grained trace model. Section 5 shows the labeled interleaving tree model, and the patterns of CDRSes. Section 6 reports the implementation and experiments about our prototype tool FGVT. Section 7 concludes the paper with future work.

2 Motivating Example

In this section, we illustrate the motivation of this work through a buggy concurrent data structure *PairSnapShot* [17].

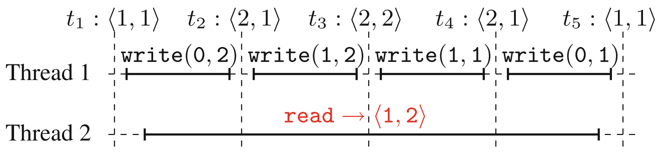
Figure 2 shows a simplified version of PairSnapShot, where it holds an array `d` of size 2. A `write(i,v)` operation writes `v` to `d[i]`, while a `read` $\rightarrow \langle v_0, v_1 \rangle$ operation reads the values of `d[0]` and `d[1]`, which are v_0 and v_1, respectively.

A correctness criterion of PairSnapShot is that `read` should always return the values of the same moment. However, Fig. 3 shows a concurrent execution in which the return values of `read` do not exist at any moment of the execution. In Fig. 3, time moves from left to right; dark lines indicate the time intervals of operations and the short vertical lines at the both ends of a dark line represent the moment when an operation is invoked and returned, respectively. A label $t : \langle v_0, v_1 \rangle$ indicates that at the moment t, `d[0]` is v_0 and `d[1]` is v_1. The operation `read` on Thread 2 returns a value $\langle 1, 2 \rangle$, which is not consistent with value of any moment.

The reason of this violation can be found out by enumerating the possible executing orders of memory access events which is labeled by `#` in Fig. 2. One possible order that can trigger the violation is illustrated in Fig. 4, in which "x" indicate the executing moments of the corresponding memory access events. Actually, this model checking approach is the most common way to locate the root cause of concurrency bugs, and has been studied in many existing literatures. Here, our focus is not on how to find this fine-grained executing order, but to study how the thread execution order, which causes data race, influences the final result of linearizability.

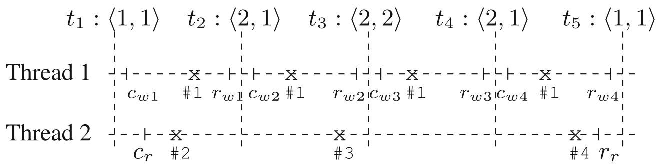
```
PairSnapShot:
    int d[2];
    write(i,v){
        d[i] = v;                   #1
    }
    Pair read(){
        while(true){
            int x = d[0];           #2
            int y = d[1];           #3
            if(x == d[0])           #4
                return <x,y>;
        }
    }
```

Fig. 2. A concurrent data structure: PairSnapShot



$t_1 : \langle 1,1 \rangle$
$t_2 : \langle 2,1 \rangle$
$t_3 : \langle 2,2 \rangle$
$t_4 : \langle 2,1 \rangle$
$t_5 : \langle 1,1 \rangle$
Thread 1
write(0, 2)
write(1, 2)
write(1, 1)
write(0, 1)
Thread 2
read $\rightarrow \langle 1,2 \rangle$

Fig. 3. A buggy trace of PairSnapShot



$t_1 : \langle 1,1 \rangle$
$t_2 : \langle 2,1 \rangle$
$t_3 : \langle 2,2 \rangle$
$t_4 : \langle 2,1 \rangle$
$t_5 : \langle 1,1 \rangle$
Thread 1
c_{w1} #1 r_{w1} c_{w2} #1 r_{w2} c_{w3} #1 r_{w3} c_{w4} #1 r_{w4}
Thread 2
c_r #2 #3 #4 r_r

Fig. 4. An executing order of memory access events triggering the violation in Fig. 3

3 Preliminary

In this section, we extend the traditional coarse-grained trace model [3], recalled in Sect. 3.1, to the fine-grained trace model presented in Sect. 3.2. Compared to the traditional one, our new model includes memory access instructions such as `read`, `write` and atomic primitive *compare-and-swap* (`CAS`). This enables us to reason about the causes of linearizability faults on the fine-grained level.

3.1 Coarse-Grained Trace Model

A *trace* S is a finite sequence of events $e(Arg)^{\langle o,t\rangle}$, where e is an event symbol ranging over a pre-defined set E, Arg represents the list of arguments, o belongs to a set O of operation identifiers and t belongs to a set T of thread identifiers. In the coarse-grained trace model, the set E contains the following subsets:

– C contains symbols that represent operation invocation events. An invocation event is represented as $c(v_a)^{\langle o,t\rangle}$ ($c \in \mathsf{C}$), where v_a is the argument of the operation;
– R contains symbols that represent operation response events. A response event is represented as $r(v_r)^{\langle o,t\rangle}$ ($r \in \mathsf{R}$), where v_r is the return value of the operation.

In this paper we also use C, R to represent the set of the corresponding events indiscriminately, and symbol $e \in \mathsf{C} \cup \mathsf{R}$ to represent an event. The order relation between events in a trace S is written as $\prec_S$ (or $\prec$), i.e., $e_1 \prec_S e_2$ if e_1 is ordered before e_2 in S. We denote the operation identifier of an event e as $\mathtt{op}(e)$, and thread identifier as $\mathtt{td}(e)$. An invocation event $c \in \mathsf{C}$ and a response event $r \in \mathsf{R}$ *match* if $\mathtt{op}(c) = \mathtt{op}(r)$, written as $c \diamond r$. A pair of matching events forms an operation instance with an operation identifier in O, and we usually represent such an operation instance as $m(v_a) \rightarrow v_r$, where m is the operation name.

A trace $S = e_1 e_2 \cdots e_n$ is *well-formed* if it satisfies that:

– Each response is preceded by a matching invocation:
  $e_j \in \mathsf{R}$ implies $e_i \diamond e_j$ for some $i < j$
– Each operation identifier is used in at most one invocation/response:
  $\mathtt{op}(e_i) = \mathtt{op}(e_j)$ and $i < j$ implies $e_i \diamond e_j$

A well-formed trace S can also be treated as a partial order set $\langle S, \sqsubset_S\rangle$ of operations on *happen-before* relation $\sqsubset_S$ between operations, where S is called a *coarse-grained trace* (or *coarse-grained history*). The *happen-before* relation $\sqsubset_S$ is defined as that: assuming two operations O_1, O_2 in S are formed by c_1, r_1 and c_2, r_2 respectively, then $O_1 \sqsubset_S O_2$ if and only if $r_1 \prec c_2$.

Example 1. Figure 3 shows such a well-formed trace: $S = c_{w1} c_r r_{w1} c_{w2} r_{w2} c_{w3} r_{w3} c_{w4} r_r r_{w4}$, where c_{wi} and r_{wi} represents the invocation and response events of the i-th `write` operation respectively, and c_r and r_r represent the invocation and response events of the `read` operation.

In this example, it is obvious that $\mathtt{write}(0,2) \sqsubset_S \mathtt{write}(1,2) \sqsubset_S \mathtt{write}(1,1) \sqsubset_S \mathtt{write}(0,1)$, but there is no *happen-before* relation between the `read` operation and any of the `write` operations.

A coarse-grained trace S is *sequential* if $\sqsubset_S$ is a total order. We define that a *specification* of an object is the set of all sequential traces that satisfy the correctness criteria of that object. Note that here correctness criterion is specified by concurrent data structures, such as *first-in-first-out* rule of *FIFO-Queue*, *first-in-last-out* rule of *Stack*.

$$\begin{array}{llllll}
1: \texttt{read} \rightarrow \langle 1,2\rangle & \texttt{write}(0,2) & \texttt{write}(1,2) & \texttt{write}(1,1) & \texttt{write}(0,1) \\
2: \texttt{write}(0,2) & \texttt{read} \rightarrow \langle 1,2\rangle & \texttt{write}(1,2) & \texttt{write}(1,1) & \texttt{write}(0,1) \\
3: \texttt{write}(0,2) & \texttt{write}(1,2) & \texttt{read} \rightarrow \langle 1,2\rangle & \texttt{write}(1,1) & \texttt{write}(0,1) \\
4: \texttt{write}(0,2) & \texttt{write}(1,2) & \texttt{write}(1,1) & \texttt{read} \rightarrow \langle 1,2\rangle & \texttt{write}(0,1) \\
5: \texttt{write}(0,2) & \texttt{write}(1,2) & \texttt{write}(1,1) & \texttt{write}(0,1) & \texttt{read} \rightarrow \langle 1,2\rangle
\end{array}$$

Fig. 5. Five possible sequential traces but none of them satisfies the criterion of PairSnapShot

Definition 1 (Linearizability). *A coarse-grained trace S of an object is linearizable if there exists a sequential trace S' in the specification of the object such that:*

1. $S = S'$, i.e., operations in S and S' are the same;
2. $\sqsubset_S \subseteq \sqsubset_{S'}$, i.e., given two operations O_1 and O_2 respectively in S and S', if $O_1 \sqsubset_S O_2$, then $O_1 \sqsubset_{S'} O_2$.

Note that this definition speaks only complete traces, neglecting the existence of pending operations, that is, the operations without response events. Since this paper focuses on analysis of linearizability faults rather than detection, we consider complete traces only.

Example 2. Figure 5 shows 5 sequential traces that satisfy requirements 1 and 2 in Definition 1 with respect to the coarse-grained trace shown in Fig. 3. However, neither of them satisfies the correctness criteria of *PairSnapShot*, by which the `read` operation should not return $\langle 1, 2\rangle$, so that neither of them belongs to the specification set of *PairSnapShot*. Therefore we can say that the coarse-grained trace in Fig. 3 is non-linearizable.

3.2 Fine-Grained Trace Model

In the fine-grained trace model, the symbol set E has other subsets of events in addition to C and R:

– Wr contains symbols that represent the *memory writing* events. An event of memory writing is represented as $wr(addr, v)^{\langle o,t\rangle}$ ($wr \in \mathsf{Wr}$), where *addr* is the memory location to be modified to a value v;
– Rd contains symbols that represent the *memory reading* events. An event of memory reading is represented as $rd(addr)^{\langle o,t\rangle}$ ($rd \in \mathsf{Rd}$), where *addr* is the memory location to be read;
– CAS contains symbols that represent some *atomic primitives*, such as *compare-and-swap* (*CAS*), in the modern architecture. A *CAS* event can be represented as $cas(addr, v_e, v_n)^{\langle o,t\rangle}$ ($cas \in \mathsf{CAS}$), where *addr* represents a memory location, v_e and v_n are two values. The functionality of this atomic primitive is that if the value at *addr* equals to v_e, it will be updated to v_n and return `true`, otherwise it would do nothing but return `false`.

Similarly, in this paper $\mathsf{Wr}, \mathsf{Rd}, \mathsf{CAS}$ also represent the corresponding sets of events. Let $\mathsf{M} = \mathsf{Wr} \cup \mathsf{Rd} \cup \mathsf{CAS}$, events e such that $e \in \mathsf{M}$ are called *memory access events.*

A fine-grained trace S_f is a total order set $\langle S_f, \prec \rangle$ of events over set $E = \mathsf{C} \cup \mathsf{R} \cup \mathsf{Wr} \cup \mathsf{Rd} \cup \mathsf{CAS}$. We define a projection $\mathcal{F}_c$ that maps a fine-grained trace S_f to a coarse-grained trace S_c by dropping all memory access events in S_f, i.e., $\mathcal{F}_c(S_f) = S_f|_{\{\mathsf{C},\mathsf{R}\}}$. A fine-grained trace S_f is *well-formed* if it satisfies that:

- $\mathcal{F}_c(S_f)$ is well-formed;
- Let $e_m(Arg)^{\langle o,t \rangle}$ be a memory access event in S_f with thread identifier o. Then o must occur in a pair of matching invocation and response events $c(Arg)^{\langle o,t \rangle}$ and $r(Arg)^{\langle o,t \rangle}$ in S_f.
- Let $c(Arg)^{\langle o,t \rangle}, r(Arg)^{\langle o,t \rangle}$ be a pair of matching invocation and response events in S_f with thread identifier o. All the memory access events e with thread identifier o in S_f should satisfy $c \prec e \prec r$.

We claim that all fine-grained traces in this paper are well-formed.

Example 3. Figure 4 presents a well-formed fine-grained trace:
$S_f = c_{w1}c_r\texttt{\#2\#1}r_{w1}c_{w2}\texttt{\#1\#3}r_{w2}c_{w3}\texttt{\#1}r_{w3}c_{w4}\texttt{\#1\#4}r_r r_{w4}$
and the $\prec$ relations between events are obvious. Application of $\mathcal{F}_c$ to this fine-grained trace results in the coarse-grained trace in Fig. 3.

Definition 2 (Linearizability of fine-grained trace). *The linearizability of S_f depends on the linearizability of $\mathcal{F}_c(S_f)$, i.e., if $\mathcal{F}_c(S_f)$ is linearizable, then S_f is linearizable.*

We define a predicate $\mathcal{L}_n$ to denote the linearizability of S_f, i.e., if S_f is linearizable, then $\mathcal{L}_n(S_f)$ is true.

4 Critical Data Race Sequence

In this section, we analyze linearizability faults on the fine-grained level, and propose *critical data race sequence* (*CDRS*) which exposes the root cause of linearizability faults.

Definition 3 (Concurrent program). *Given a coarse-grained trace S_c, we define a concurrent program $\mathbb{P}$: $\mathbb{P}$ is a set of fine-grained traces such that every fine-grained trace $S_f \in \mathbb{P}$ can be mapped to a coarse-grained trace S'_c, which satisfies that:*

- *$S'_c = S_c$, i.e., operations in S'_c and S_c are the same;*
- *$\sqsubset_{S_c} \subseteq \sqsubset_{S'_c}$, i.e., S'_c preserves the happen-before relation in S_c.*

Intuitively, a program $\mathbb{P}$ maintains all fine-grained traces that preserve the *happen-before* relation of S_c. If S_c is sequential, then we say the program $\mathbb{P}$ w.r.t. S_c is sequential. And, if there exists a non-linearizable fine-grained trace in a program $\mathbb{P}$, we say that $\mathbb{P}$ is non-linearizable.

Definition 4 (Linearizability fault). *Let $\mathbb{P}$ be a non-linearizable concurrent program. A linearizability fault $\mathfrak{F}$ is defined as a non-linearizable fine-grained trace S_f in $\mathbb{P}$.*

We define the prefix relation $\subseteq_{pre}$ between two fine-grained trace S_1 and S_2, that is, $S_1 \subseteq_{pre} S_2$ says that S_1 is a prefix of S_2. We use $\cdot$ to represent the concatenation of a sequence of events and another sequence of events, that is, if $S_1 = e_1 \cdots e_n$ and $S_2 = e_{n+1} \cdots e_{n+m}$, then $S_1 \cdot S_2 = e_1 \cdots e_n e_{n+1} \cdots e_{n+m}$.

Definition 5 (High-level data race [1]). *Let $\mathbb{P}$ be a concurrent program. A high-level data race (HLDR) D in $\mathbb{P}$ is defined as a triple $\langle Var, I_e, M_e \rangle$. Here, Var is a set of one or more shared variables, each corresponding to a memory location. I_e is a sequence of events. M_e is a set of two or more memory access events $e(Arg)^{\langle o,t \rangle}$, such that:*

- *each event e has a distinct thread identifier o;*
- *each event e accesses some shared variables in Var;*
- *for any permutation S_p of events in M_e, there exists a fine-grained trace $S \in \mathbb{P}$ such that $I_e \cdot S_p \subseteq_{pre} S$.*

Note that here the elements of *Var* depends on the algorithm of the object. The most common situation is that *Var* contains one shared variable which is accessed by several events simultaneously. However, there also exist other situations, such as *PairSnapShot* in Sect. 2, in which several memory locations should be considered globally.

Given a HLDR $\mathsf{D} = \langle Var, I_e, M_e \rangle$ where $e_1, e_2 \in M_e$, we say e_1 *wins* e_2 with respect to a fine-grained trace S_f if $I_e \cdot e_1 e_2 \subseteq_{pre} S_f$.

Given a program $\mathbb{P}$, we define a partial order relation $<_{dr}$ between two HLDRs $\mathsf{D}_1 = \langle Var_1, I_{e1}, M_{e1} \rangle$ and $\mathsf{D}_2 = \langle Var_2, I_{e2}, M_{e2} \rangle$ as that if $I_{e1} \subseteq_{pre} I_{e2}$, then $\mathsf{D}_1 <_{dr} \mathsf{D}_2$.

Theorem 1. *If there is a linearizability fault, then there is a high-level data race.*

Proof. To prove Theorem 1, it suffices to prove the contrapositive proposition that if there is no high-level data race, then there is no linearizability fault. According to Definition 5, the premise, no high-level data race, means that in M_e:

$$\exists S_p \forall S (I_e \cdot S_p \nsubseteq_{pre} S)$$

Here, if $\mathbb{P}$ is a concurrent program, this condition is not satisfied according to Definition 3. Therefore, the $\mathbb{P}$ that satisfies this condition corresponds to a sequential program, and the sequential trace surely has no linearizability fault.

Definition 6 (Critical data race sequence). *Let $\mathbb{P}$ be a concurrent program, $\mathfrak{F}$ be a linearizability fault. A Critical Data Race Sequence (CDRS) with respect to $\mathfrak{F}$ is a total order set of data races* $\{\mathsf{D}_1, \mathsf{D}_2, \cdots, \mathsf{D}_n\} =$

$$\left\{\begin{array}{c}\langle Var_1, I_{e1}, M_{e1} = \{e_{11}, e_{12}, \cdots, e_{1m_1}\}\rangle, \\ \langle Var_2, I_{e2}, M_{e2} = \{e_{21}, e_{22}, \cdots, e_{2m_2}\}\rangle, \\ \vdots \\ \langle Var_n, I_{en}, M_{en} = \{e_{n1}, e_{n2}, \cdots, e_{nm_n}\}\rangle\end{array}\right\}$$

where the relation $<_{dr}$ *exists as* $\mathsf{D}_1 <_{dr} \mathsf{D}_2 <_{dr} \cdots <_{dr} \mathsf{D}_n$. *A CDRS satisfies that there exist two events* $e_{i1}, e_{i2} \in M_{ei}$ $(i \in 1, \ldots, n)$ *that* e_{i1}*'s win and* e_{i2}*'s win lead the program to "inverse" consequences. Here, "inverse" includes two cases:*

- *If all the fine-grained traces* S_{f1} *satisfying* $I_{ei} \cdot e_{i1} \subseteq_{pre} S_{f1}$ *are linearizable, then there exist non-linearizable fine-grained traces* S_{f2} *satisfying* $I_{e\,i} \cdot e_{i2} \subseteq_{pre} S_{f2}$*;*
- *If all the fine-grained traces* S_{f1} *satisfying* $I_{ei} \cdot e_{i1} \subseteq_{pre} S_{f1}$ *are non-linearizable, then there exist linearizable fine-grained traces* S_{f2} *satisfying* $I_{e\,i} \cdot e_{i2} \subseteq_{pre} S_{f2}$.

Intuitively, a *CDRS* contains all the HLDRs which are decisive to the linearizability of the trace. Note that although different *CDRS*es in a program $\mathbb{P}$ may lead to different linearizability faults, we only focus on the consequence that linearizability is violated and thus we treat all linearizability faults identical in the sense that they all lead the trace non-linearizable.

Example 4. Take a look at the example of a HLDR $\mathsf{D} = \langle Var, I_e, M_e\rangle$ in *PairSnapShot* in which $M_e = \{\texttt{\#1,\#2}\}$. If `#1` wins, then a non-linearizable trace will never occur; but if `#2` wins like Fig. 4, there exists at least one such fine-grained trace that is non-linearizable. In this case, D is included in a CDRS with respect to the linearizability fault $\mathfrak{F}$ shown in Fig. 4.

5 Identify CDRS on Interleaving Tree

From Sect. 4, we know that it is the competitions happening in *CDRS*es that trigger the linearizability faults. In order to identify *CDRS*, we propose an approach based on a model called labeled interleaving tree in this section. Firstly, we express the fine-grained traces in an interleaving tree, and then we label the nodes of the tree with a symbol system. We will show that all *CDRS*es follow a certain pattern and thus we can identify them based on the characteristics of the nodes.

5.1 Interleaving Tree

Firstly, we define a projection $\mathcal{F}_M$ mapping a fine-grained trace S_f to a trace S_f^M composed of only memory access events in S_f, i.e., $\mathcal{F}_M(S_f) = S_f|_{\mathsf{M}}$. The linearizability of S_f^M is decided by that of S_f, i.e., $\mathcal{L}_n(S_f^M) = \mathcal{L}_n(S_f)$. We define a state $\mathcal{S}_t$ of an object to be a map from a memory locations to its value, e.g., in Fig. 3, $\mathcal{S}_t(\mathsf{d}[0]) = 1$, $\mathcal{S}_t(\mathsf{d}[1]) = 1$ at $\mathtt{t}_1$.

Definition 7 (Interleaving Tree). *An Interleaving Tree of $\mathbb{P}$ is a tree, where each node corresponds to a state, and each edge corresponds to a memory access event. A subtree rooted at node N_d is represented as $\mathcal{T}ree(N_d)$. The set of the leaves of the tree is represented as N_{lf}.*

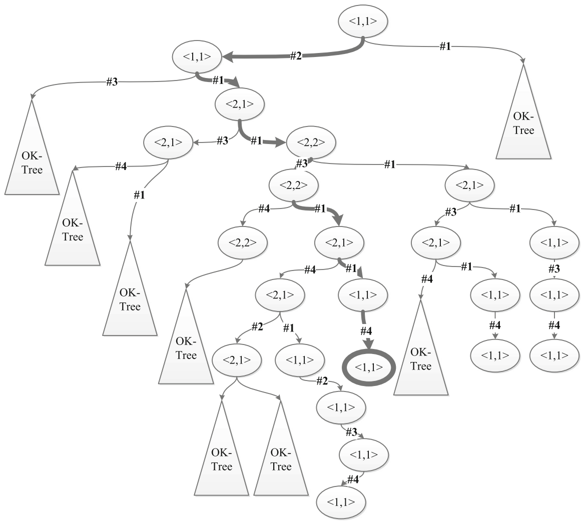


Fig. 6. Interleaving tree of PairSnapShot

Algorithm 1 presents how to build an interleaving tree recursively. In line 1, $\mathcal{S}_t$ is initialized to the initial state of the object. The set *enS* in line 2 initially contains the events which are minimal w.r.t. $\prec$ over the events with the same thread identifier, and thus the number of elements in *enS* is the same as the number of threads. Line 3–11 is the process of building the tree. Firstly, a node is built as line 4 shows. Then, events in *enS* are traversed, each accessing a memory location *addr* as line 5 shows. When an event is accessed, a corresponding edge is built as in line 6. Then the state is updated by substituting the value of *addr* with v_n as in line 7. Here note that if e is an Rd event, $\mathcal{S}_t$ will not be modified. And *enS* is updated as line 8 shows, where e will be replaced by its successor w.r.t. $\prec$ over the events with the same thread identifier. Finally, the updated $\mathcal{S}_t$ and *enS* are applied as arguments to another invocation of BUILDTREE as line 9 shows to build a subtree recursively.

Example 5. According to Algorithm 1, we build the interleaving tree of the concurrent program $\mathbb{P}$ corresponding to Fig. 3 and present it in Fig. 6. Each node represents a state, and each edge represents a memory access event.

Although due to the limitation of space we have omitted many paths, we can still see that this tree maintains all the fine-grained traces in $\mathbb{P}$, and among these traces the one with bold paths corresponds to the non-linearizability situation in Fig. 4.

Algorithm 1. Building of Interleaving Tree

```
1: $\mathcal{S}_t = \mathcal{S}_t^{init}$                                  ▷ $\mathcal{S}_t$ is initialized
2: $enS = \{e | \forall \epsilon \in \mathsf{M}.(\mathtt{td}(\epsilon) = \mathtt{td}(e) \longrightarrow e \prec \epsilon)\}$   ▷ Foremost events of each thread
3: function BUILDTREE($\mathcal{S}_t$, $enS$)
4:     NEW NODE($\mathcal{S}_t$)
5:     for $e(addr, v_n)^{\langle o,t \rangle} \leftarrow enS$ do
6:         NEW EDGE($e$)
7:         $\mathcal{S}_t \leftarrow \mathcal{S}_t[v_n/addr]$                    ▷ Update state
8:         $enS \leftarrow enS \setminus \{e\} \cup \{e'\}$                 ▷ Update $enS$
9:         BUILDTREE($\mathcal{S}_t$, $enS$)              ▷ Recursively build the tree
10:    end for
11: end function
```

5.2 Identify CDRS on Labeled Interleaving Tree

After building an interleaving tree, we design a symbol system to label the tree in order for the identification of CDRS.

Since each leaf $l_f \in N_{lf}$ corresponds to a fine-grained trace from root to itself, we directly apply $\mathcal{L}_n$ to l_f to check the linearizability of the corresponding fine-grained trace. Here, we take a binary interleaving tree built by traces with two threads to illustrate our symbol system.

In our system, a subtree $\mathcal{Tree}(N_d)$ rooted at N_d and holding a leaf set N_{lf} can be grouped into one of the following categories:

- *OK*-tree — all fine-grained traces are linearizable,
  $\forall l_f \, (l_f \in N_{lf} \rightarrow \mathcal{L}_n(l_f))$
- *ERR*-tree — all fine-grained traces are non-linearizable,
  $\forall l_f \, (l_f \in N_{lf} \rightarrow \neg \mathcal{L}_n(l_f))$
- *MIX*-tree — both linearizable and non-linearizable fine-grained traces exist,
  $\exists l_{f1}(l_{f1} \in N_{lf} \wedge \mathcal{L}_n(l_{f1})) \wedge \exists l_{f2}(l_{f2} \in N_{lf} \wedge \neg \mathcal{L}_n(l_{f2}))$

Definition 8 (Node Labeling). *Based on the categories of subtrees, a node N_d can be labeled as one of the following symbols,*

- *W-node — if $\mathcal{Tree}(N_d)$ is an OK-tree.*
- *B-node — if $\mathcal{Tree}(N_d)$ is an ERR-tree.*
- *G-node — if one subtree of N_d is an OK-tree, and the other is an ERR-tree.*
- *GG-node — if two subtrees of N_d are both MIX-trees.*
- *WG-node — if one subtree of N_d is an OK-tree, and the other is a MIX-tree.*
- *BG-node — if one subtree of N_d is an ERR-tree, and the other is a MIX-tree.*

where W represents white, B represents black and G represents grey actually. Figure 7 illustrates this labeling rule.

The algorithm of labeling an interleaving tree is presented in Algorithm 2. The function LABELNODE recursively labels the nodes of a tree. Firstly, it checks whether the node being labeled has left or right child in line {3, 6, 8, 10}, where

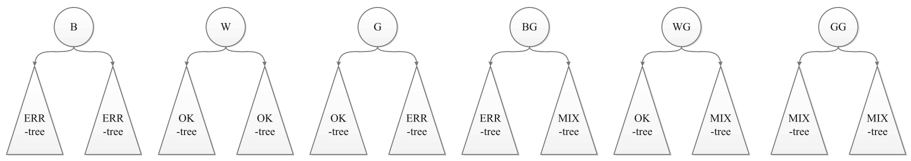


Fig. 7. Labels of nodes

`Left`(N_d) gets the left child of N_d, and `Right`(N_d) gets the right child. !`Left`(N_d) means that N_d has no left child, and !`Right`(N_d) is in a similar way. So if both !`Left`(N_d) and !`Right`(N_d) are true, it means N_d is a leaf and thus it is labeled depending on the linearizability of itself as line 3–5 shows. Otherwise, the node is labeled depending its left and right child as line 6–27 shows.

Theorem 2 (Completeness). *Each node of the interleaving tree belongs to one kind of the nodes in Definition 8.*

Actually, each node N_d of an interleaving tree together with all of its out-edges corresponds to a data race $\mathsf{D} = \langle Var, I_e, M_e \rangle$. The set Var is a subset of the domain of $\mathcal{S}_t$, where $\mathcal{S}_t$ is represented by the value in a node N_d (e.g., Fig. 6), I_e is a prefix composed of events represented by edges from the root to N_d, and M_e contains all events e each corresponding to an out-edge of N_d. Therefore, we can uniquely identify a data race by a node.

Theorem 3 (Identifying CDRS). *A CDRS is equivalent to a subset of nodes in a root-to-leaf path, satisfying a regular expression form*

$$(W_g|B_g)^*(B_g|G)$$

where W_g, B_g, G respectively represent WG-node, BG-node, G-node.

Proof.
- Firstly we show that the node sequence following $(W_g|B_g)^*(B_g|G)$ in an interleaving tree is a CDRS. From the definition of W_g-node, B_g-node, and G-node, it is obvious that the HLDRs composed by these 3 kinds of node and their out-edges all belong to the data races described in the Definition 6.
- Then we show that a CDRS appears as $(W_g|B_g)^*(B_g|G)$ in an interleaving tree. According to Definition 6, the two different cases for "inverse" consequences correspond to the W_g-node and B_g-node. Furthermore, since CDRS implies a linearizability fault $\mathfrak{F}$, the ending of a CDRS should be that there exists an event whose win can lead all fine-grained trace non-linearizable, and that is just the case of BG-node and G-node, which corresponds to the expression in the theorem.

Example 6. Take a look at Fig. 6. We label the tree according to Definition 8, and get a labeled tree in Fig. 8. As we can see, the thickened path with a red leaf is non-linearizable, the nodes on which include a CDRS. The CDRS is shown by the sequence of yellow nodes, in the form of $W_g\,W_g\,W_g\,W_g\,W_g\,G$, which is accepted by the regular expression in Theorem 3.

Algorithm 2. Labeling Interleaving Tree

```
$Label = \{W, B, G, GG, WG, BG\}$
function LabelNode($N_d$)
    if !Left($N_d$)&!Right($N_d$) then                        ▷ $N_d$ is a leaf
        if $\mathcal{L}_n(N_d)$ then return $W$
        else return $B$
    else if Left($N_d$)&!Right($N_d$) then                     ▷ $N_d$ has only left child
        return LabelNode(Left($N_d$))
    else if !Left($N_d$)&Right($N_d$) then                     ▷ $N_d$ has only right child
        return LabelNode(Right($N_d$))
    else                                                       ▷ $N_d$ has both children
        $Label_l$ = LabelNode(Left($N_d$))
        $Label_r$ = LabelNode(Right($N_d$))
        switch $\langle Label_l, Label_r\rangle$                    ▷ The label depends on 2 children
            case $\langle W, W\rangle$:
                return $W$
            case $\langle B, B\rangle$:
                return $B$
            case $\langle B, W\rangle|\langle W, B\rangle$
                return $G$
            case $\langle (B|W|G)?G, W\rangle|\langle W, (B|W|G)?G\rangle$:
                return $WG$
            case $\langle (B|W|G)?G, B\rangle|\langle B, (B|W|G)?G\rangle$:
                return $BG$
            case $\langle (B|W|G)?G, (B|W|G)?G\rangle$:
                return $GG$
        end switch
    end if
end function
```

6 Implementation and Evaluation

We have integrated what we presented in Sect. 5 into a prototype tool called FGVT (Fine-grained VeriTrace), and experiments show that given a *minimum test case* [29], our tool is able to localize the CDRS. In this section, we will give a brief introduction about our tool and experiments, and display the experiment results to show the power of FGVT.

6.1 Implementation

Our tool FGVT is based on the framework of JavaPathFinder (JPF), which encapsulates a Java virtual machine and can be customized for use of model checking of Java programs. JPF is employed to generate interleaving trees, and it applies a dynamic reduction mechanism to eliminate duplicated program states and simplify the interleaving tree. Then, we label the tree based on Algorithm 2, and report the CDRSes which cause linearizability faults.

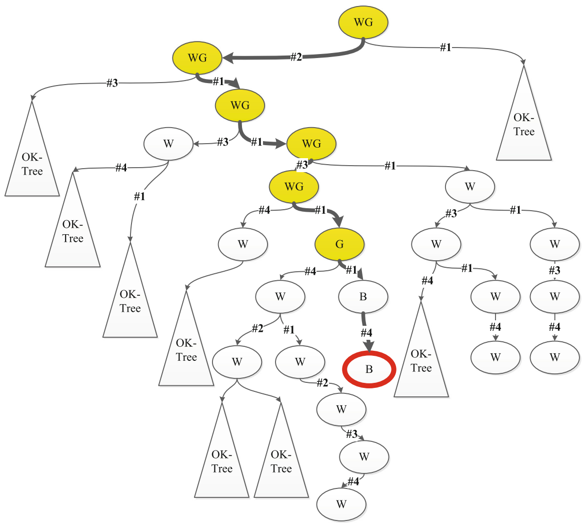


Fig. 8. Labeled interleaving tree

Table 2. Evaluation result

Concur. object	Initial State	Operations	CDRS	Relating data race

LockFreeList	{1}	thd1:`remove`(1)
		thd2:`remove`(1)
			W_gG	`curr.next.get()`
				`attemptMark()`

OptimisticQueue	{1,2}	thd1:`poll`()
		thd2:`poll`()
			G	`head.getItem()`
				`casHead()`

PairSnapshot	$\langle 1, 1\rangle$	thd1: `write`(0, 2),
		`write`(1, 2),
		`write`(1, 1),
		`write`(0, 1)
		thd2:`read`()
			$W_g\{5\}G$	`d[i]=v`
				`x=d[0],y=d[1]`
				`if(x==d[0])`

Snark	{1}	thd1:`popRight`()
		thd2:`pushRight`(2),
		`popLeft`()
			W_gW_gG	`rh=RightHat`
				`DCAS(&RightHat,...)`
				`DCAS(&LeftHat,...)`
				`if(rh.R==rh)`

SimpleList	{1}	thd1:`add`(3)
		thd2:`add`(4)
			B_g or B_gG	`pred.next=node`
				`curr.val<v`

LinkedList	{1,5}	thd1:`remove`(1),
		`add`(9)
		thd2:`size`()
			W_gW_gG	`node.next==tail`
				`synchronized(){...}`

6.2 Benchmark

In this section, we will introduce some concurrent objects in our experiment. In addition to PairSnapShot, we also do experiments on many other concurrent objects, which have been proved to be non-linearizable by other tools.

- LockFreeList [11] — It is a concurrent Set that violates linearizability when two `removes` compete to mark a bit without synchronization protection.
- OptimisticQueue [15] — It is a concurrent Queue that violates linearizability when two `poll` operations compete to get the head of the queue. Without proper synchronization between reading `head` pointer and modifying it, two `polls` may return the same value.
- Snark [6] — It is a Deque with the use of DCAS (*double-compare-and-swap*), and violates linearizability when the object has few elements and operations originally accessing different ends compete for the same memory location.
- SimpleList [27] — It is a concurrent Set and the bug is typical. The `Add` function inserts a node by modifying the `next` pointer of its predecessor, but without protection, `next` may be modified by other threads leading the node removed from the list unexpectedly.
- Operation `size` of Linked List — As we know, `size` is used for counting the number of nodes in a list. However, if there is no synchronization, a situation that violates linearizability happens when `size` traverses the list, another thread preempts the execution and deletes a node which has been accessed and inserts a node at a position that has not been accessed, so `size` will return a value that is larger than the expected length.

6.3 Evaluation

We evaluate our tool by 6 test cases either from prior work or from real applications as Sect. 6.2 introduced. In our experiments, all the concurrent objects are executed by two threads, and the operations being tested, initial states of each concurrent object and arguments are listed in Columns 2–3 of Table 2.

The node sequence patterns found based on the labeled interleaving tree are listed in Column 4 of Table 2. As we expect, these patterns exactly correspond to the CDRSes of each test case, and here we got some conclusions from the experimental results:

- All the patterns follow the form of regular expression in Theorem 3.
- Most of the test cases end with a G-node, and *SimpleList* shows us a sequence ending with a BG-node.
- We can see the case where not only one CDRS exists.

In this experiments, the executing time for each example is acceptable. That is because the inputs for the experiments are all small traces containing less than 5 operations obtained by the coarse-grained localization tool [29] as we introduced in Sect. 1, though such tree structures suffer from the state space explosion problem.

Column 5 lists the source code in the original program that causes the data races in the CDRSes. The reason we project the CDRSes to the source code is that we try to locate the critical data races on the code level and find approaches to repair them. For example, we can repair the linearizability faults in *Lock-FreeList* by transforming `attemptMark` into `compareAndSet`. However, this is not always feasible such as the situation in *PairSnapShot*, where we cannot point out exactly modifying which lines can lead the object linearizable. Despite such situations, our approach manages to give helpful information on the location of critical data races that triggering linearizability faults and further facilitate bug repair.

7 Conclusion

This paper proposes the notion of *critical data race sequence* (*CDRS*) that characterizes the root causes of linearizability faults based on a fine-grained trace model. A CDRS is a set of data races that are decisive to trigger linearizability faults. Therefore, the existence of a CDRS implies that a concurrent execution has potential to be non-linearizable. We also present a labeled interleaving tree model to support automated identification of CDRS. A tool called FGVT is then developed to automatically identify CDRSes and localize the causes of linearizability faults. Experiments have well demonstrated its effectiveness and efficiency.

This work reveals the pattern of the data races that are decisive on the linearizability of a trace. These data races can be mapped to certain parts of the source code. In the future work, it would be interesting to establish a stronger relationship between CDRSes and the source code for the sake of bug analysis and repair.

Acknowledgements. We thank the anonymous referees for their valuable comments. This work is partially supported by the National Key Basic Research Program of China under the Grant No. 2014CB340701, the National Key Research and Development Program of China under the Grant No. 2017YFB0801900, the CAS-INRIA Joint Research Program under the Grant No. GJHZ1844, and the ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST.

References

1. Artho, C., Havelund, K., Biere, A.: High-level data races. Softw. Test. Verif. Reliab. **13**(4), 207–227 (2003)
2. Ben-Asher, Y., Farchi, E., Eytani, Y.: Heuristics for finding concurrent bugs. In: International Symposium on Parallel and Distributed Processing, p. 288a (2003)
3. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Tractable refinement checking for concurrent objects. In: Rajamani, S.K., Walker, D. (eds.) Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, 15–17 January 2015, pp. 651–662. ACM (2015)

4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: Zorn, B.G., Aiken, A. (eds.) Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, 5–10 June 2010, pp. 330–340. ACM (2010)
5. Choi, J.D., Srinivasan, H.: Deterministic replay of java multithreaded applications. In: Proceedings of the Sigmetrics Symposium on Parallel & Distributed Tools, pp. 48–59 (2000)
6. Doherty, S., et al.: DCAS is not a silver bullet for nonblocking algorithm design. In: Gibbons, P.B., Adler, M. (eds.) Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2004, 27–30 June 2004, Barcelona, Spain, pp. 216–224. ACM (2004)
7. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded Java programs. Concurr. Comput.: Pract. Exp. **15**(3–5), 485–499 (2003)
8. Emmi, M., Enea, C., Hamza, J.: Monitoring refinement via symbolic reasoning. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 15–17 June 2015, pp. 260–269. ACM (2015)
9. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22–26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings, p. 286. IEEE Computer Society (2003)
10. Gottschlich, J.E., Pokam, G., Pereira, C., Wu, Y.: Concurrent predicates: a debugging technique for every parallel programmer. In: Fensch, C., O'Boyle, M.F.P., Seznec, A., Bodin, F. (eds.) Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, United Kingdom, 7–11 September 2013, pp. 331–340. IEEE Computer Society (2013)
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier (2012)
12. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
13. Horn, A., Kroening, D.: Faster linearizability checking via P-compositionality. In: Graf, S., Viswanathan, M. (eds.) FORTE 2015. LNCS, vol. 9039, pp. 50–65. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19195-9_4
14. Khoshnood, S., Kusano, M., Wang, C.: Concbugassist: constraint solving for diagnosis and repair of concurrency bugs. In: Young, M., Xie, T. (eds.) Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, 12–17 July 2015, pp. 165–176. ACM (2015)
15. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free FIFO queues. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 117–131. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30186-8_9
16. Liu, B., Qi, Z., Wang, B., Ma, R.: Pinso: precise isolation of concurrency bugs via delta triaging. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014, pp. 201–210. IEEE Computer Society (2014)
17. Long, Z., Zhang, Y.: Checking linearizability with fine-grained traces. In: Ossowski, S. (ed.) Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 4–8 April 2016, pp. 1394–1400. ACM (2016)
18. Lowe, G.: Testing for linearizability. Concurr. Comput.: Pract. Exp. **29**(4), e3928 (2017)

19. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Eggers, S.J., Larus, J.R. (eds.) Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, 1–5 March 2008, pp. 329–339. ACM (2008)
20. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, 10–13 June 2007, pp. 446–455. ACM (2007)
21. Park, C., Sen, K.: Concurrent breakpoints. In: Ramanujam, J., Sadayappan, P. (eds.) Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, 25–29 February 2012, pp. 331–332. ACM (2012)
22. Park, S.: Fault comprehension for concurrent programs. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, 18–26 May 2013, pp. 1444–1446. IEEE Computer Society (2013)
23. Park, S., Vuduc, R.W., Harrold, M.J.: Falcon: fault localization in concurrent programs. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1–8 May 2010, pp. 245–254. ACM (2010)
24. Park, S., Vuduc, R.W., Harrold, M.J.: A unified approach for localizing non-deadlock concurrency bugs. In: Antoniol, G., Bertolino, A., Labiche, Y. (eds.) Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, 17–21 April 2012, pp. 51–60. IEEE Computer Society (2012)
25. Shi, Y., et al.: Do I use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, 17–21 October 2010, Reno/Tahoe, Nevada, USA, pp. 160–174. ACM (2010)
26. Stoller, S.D.: Testing concurrent java programs using randomized scheduling. Electr. Not. Theor. Comput. Sci. **70**(4), 142–157 (2002)
27. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008, pp. 125–135. ACM (2008)
28. Wing, J.M., Gong, C.: Testing and verifying concurrent objects. J. Parallel Distrib. Comput. **17**(1–2), 164–182 (1993)
29. Zhang, Z., Wu, P., Zhang, Y.: Localization of linearizability faults on the coarse-grained level. In: He, X. (ed.) The 29th International Conference on Software Engineering and Knowledge Engineering, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, 5–7 July 2017, pp. 272–277. KSI Research Inc. and Knowledge Systems Institute Graduate School (2017)

Miscellaneous (Short Papers)

Improvement in JavaMOP by Simplifying Büchi Automaton

Junyan Qian(✉), Cong Chen, Wei Cao, Zhongyi Zhai, and Lingzhong Zhao

Guangxi Key Laboratory of Trusted Software,
Guilin University of Electronic Technology, Guilin 541004, China
qjy2000@gmail.com, chen362076297@gmail.com, swuncv2015@gmail.com,
zhaizhongyi@guet.edu.cn, zhaolingzhong@126.com

Abstract. Runtime verification is a lightweight verification structure with the advantages of formal verification and software testing. In this paper, we studied JavaMOP that is an instance of the MOP (monitoring oriented programming) runtime verification structure. In the LTL plugin of JavaMOP, the LTL formula was translated into a nondeterministic büchi automaton (NBA) and then the NBA was translated into a deterministic finite automaton (DFA). While in the translation process, the number of states of the automata could increase exponentially. So we first use a procedure to remove the redundant states in the NBA, then we use an algorithm based on *fair simulation* which was used in model checking to further simplify the NBA. Experimental results show that the memory usage at runtime of JavaMOP can be reduced.

Keywords: Runtime verification · LTL · *Fair simulation*
Nondeterministic büchi automaton

1 Introduction

Software system plays an important role in modern society. It becomes more and more important to improve the reliability, security and credibility of software system. While exhaustive validation of a system is often not suitable for the verification, because of the dynamic loading of some additional libraries. In this case, runtime verification [1] which is a new direction of verification and provides an alternative lightweight framework for program verification. Runtime verification is an effective technique to ensure at execution time that a system meets a desirable behavior. It can be used in numerous application domains.

Runtime monitoring of properties can increase the reliability and stability in the software systems. So there is increasing interest in monitoring in software development and analysis. Many approaches have been proposed in [2]. Various runtime verification techniques and tools such as Tracematches [3], PQL [4], JavaMOP [5,6], etc., have been developed in recent years. RV-Android [7] is a tool which is designed by Daian et al., it is used for monitoring formal safety properties on Android. Yamagata et al. [8] developed a runtime monitoring framework for concurrent system.

A method of constructing the monitor based on Büchi automata was proposed in [9]. Then this method becomes a major monitor construction method. In model checking, temporal logic specifications are usually converted into büchi automata. While in runtime verification, it is different from model checking that the temporal logic specifications are converted into DFA. In JavaMOP [10] which is a runtime verification tool, an algorithm in [11] was used to translate the LTL specifications into a NBA. But this algorithm can't ensure that the size of the NBA is optimal.

In this paper, we present an algorithm to simplify the NBA. Through using this algorithm, we can reduce the number of states in the NBA and then can reduce the size of the monitor in JavaMOP. In detail, we have provided the following major contributions:

– We first use a procedure to determine whether the accepting language of the NBA starting with a state is empty. Then, we mark these states that accepting language of the NBA is empty with a redundant flag, and we remove some states which are marked with redundant flag.
– After removing the redundant states, we use an algorithm proposed in [12] which is based on *fair simulation* [16] to further simplify the NBA.
– Finally, the translation from the NBA to nondeterministic finite automaton (NFA) and then from the NFA to DFA can be accelerated. And the memory usage in JavaMOP can also be reduced.

The rest of the paper is organized as follows. The next section recalls the definition of LTL and original transition processes of LTL plugin in the JavaMop. Section 3 focuses on new translation algorithm. Section 4 is devoted to experimental results. The last section provides some concluding remarks.

2 Preliminary

2.1 Linear Temporal Logic (LTL)

We define the set of LTL formulae by using the atomic proposition p and some temporal operators such as $\vee$ (disjunction), X (next), U (until), and $\neg$ (negation). The semantics of LTL can be seen in [13].

Definition 1 (Syntax of LTL). *Let AP be a finite and non-empty set of atomic propositions, p be an atomic proposition in a finite set of atomic propositions AP. The set of LTL formulae could be defined inductively by the following grammar:*
$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi U \varphi \mid X\varphi$

2.2 JavaMOP

JavaMOP [10] is a MOP [14] development tool for Java. In JavaMOP, there are many plugins which are used to translate the specifications into monitors, for example, LTL, ERE and CFG, et al.. In this paper, we focus on LTL plugin which deals with the specified properties by using LTL. In runtime verification, when we use the LTL to specify the property, the LTL formula will be translated into DFA. There are five steps in the translation process.

- Translating the LTL formula into an alternating automaton.
- Translating the alternating automaton into a generalized büchi automaton.
- The generalized automaton was translated into a NBA.
- Translating the NBA to a NFA.
- Translating the NFA to a DFA.

In this translation process, a LTL formula was translated into a NBA first. The algorithm used in this phase was proposed in [11]. When translating the LTL formula into a NBA, the size of the NBA was not optimal and it still can be simplified. However, translating the NFA to a DFA, we use the powerset construction algorithm which will lead to the size of the automata growing exponentially. So we need to reduce the size of the DFA. And we add an algorithm to simplify the NBA.

3 Improvement in JavaMOP

In this section, we will introduce a translation algorithm that is used to simplify the NBA. This simplification algorithm has been used in model checking, for the purpose of state space reduction. In this paper, we add this simplification algorithm in JavaMOP to simplify intermediate states during execution. And we use the algorithm to reduce the size of the NBA in JavaMOP so that improving verification efficiency. Therefore, we have a new process for translating LTL formula into DFA as shown in Fig. 1.

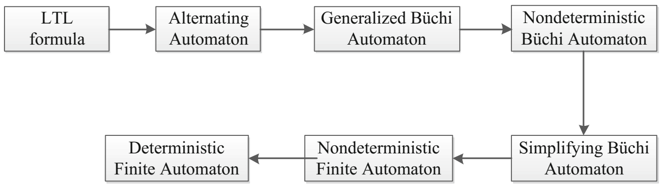

LTL formula
Alternating Automaton
Generalized Büchi Automaton
Nondeterministic Büchi Automaton
Simplifying Büchi Automaton
Nondeterministic Finite Automaton
Deterministic Finite Automaton

Fig. 1. The new process of translation from LTL formula to the deterministic finite automaton

In this algorithm, we first remove the redundant states in the NBA. For this purpose, we use an algorithm (denoted as NDFS) from [15] to identify a strong connected component in $\mathcal{B}$(is a NBA). For every state s in the büchi automaton, we first determine whether the accepting language of the automaton starting with s is empty. If it is empty, then we mark the state with a redundant flag. And then the state will be removed from the set of states Q. The detail of the algorithm is shown in Algorithm 1.

Algorithm 1. RRS(s)

```
    /*remove the redundant state form büchi automaton.*/
Input: : a nondeterministic büchi automaton $\mathcal{B} = (\Sigma, Q, Q_0, \delta, F)$
Output: : a nondeterministic büchi automaton $\mathcal{B}'$ with less states
 1: for $s \in Q$ do
 2:     if NDFS($s$) does not report cycle then
 3:         mark $s$ with the redundant flag
 4:     end if
 5: end for
 6: for $s \in Q$ do
 7:     if $s$ is marked with redundant flag then
 8:         $Q := Q \setminus \{s\}$
 9:     end if
10: end for
```

After removing the redundant states of the NBA, then we use an algorithm from [12]. The algorithm first computes the fair simulation relation by using an algorithm *Enhanced version of Jurdzinskis algorithm* from [16]. Then we can get the set of the state pairs simulate each other and the set of some transitions that would be removed. The algorithm will check whether the accepting language of the automaton $\mathcal{B}$ changes after merging these state pairs and removing these transitions. If the accepting language of $\mathcal{B}$ is not changed, and this procedure will simplify $\mathcal{B}$.

4 Experimental Results

We have implemented the algorithm in Java. A Linux PC with an Inter(R) Core(TM) i5-4690 3.50 GHz CPU and 8.0 GB of memory was used to run the experiments. For the Comparison with JavaMOP, we compare the overhead of our algorithm with that of JavaMOP. We choose JavaMOP, because it is the most efficient runtime verification tool, when we write this paper. We collected the execution time and memory usage for the benchmark, as shown in Table 1, under a steady state using DaCapo [17] 9.12's. We took the average performance of five executions. *improvement* stands for the improvement over JavaMOP in terms of the consumption of memory in Table 1. It is calculated by:

$$(1 - \frac{mem\_N}{mem}) \times 100\%$$

From Table 1 we can see that our new translation algorithm performs better than JavaMOP for the consumption of memory, because we simplified the NBA so that the size of the DFA was reduced and thus cause a reduction of the memory usage. For the property ***HasNext***, the memory used in JavaMOP is 632.19 MB, while it is 618.59 MB in ours. The improvement is 2.15%. For the property ***SafeSyncCollection***, the memory used in JavaMOP is 1002.44 MB, while it is 952.22 MB in ours. The improvement is 5.01% which is higher than

Table 1. The runtime performance comparison of JavaMOP and JM_FS

Property	Performance				*Improvement*
	JavaMOP		JM_FS		
	times (s)	*mem* (MB)	times (s)	*mem_N* (MB)	
HasNext	93.75	632.19	99.85	618.59	2.15
SafeSyncCollection	164.88	1002.44	173.88	952.22	5.01
UnsafeIterator	235.31	799.37	238.49	768.27	3.89
UnsafeMapIterator	225.61	766.05	229.91	737.48	3.73

2.15%, because this property is more complicated than ***HasNext*** and the simplification algorithm could remove more states in the NBA. The running time of our new translation algorithm is longer than JavaMOP, because we added a new procedure to the original translation process. So it causes some time consumption to run the simplification algorithm.

5 Conclusion

In this paper, we introduce the runtime verification, then we also introduce the MOP framework and JavaMOP. For the purpose of improving the performance of JavaMOP, we added a simplifying procedure to reduce the NBA which is an intermediate in the translation of a LTL formula to a DFA. After the processing of the simplification procedure, the runtime memory of JavaMOP also can be reduced. In the future, we will consider to use a more efficient simplifying algorithm which has less time and memory consumption, or we can use a more efficient translation algorithm which is used to translate a LTL formula to a NBA.

Acknowledgments. This work is supported by the National Natural Science Foundation of China under grant No. 61562015 and 61572146, the High Level Innovation Team of Guangxi Colleges and Universities and Outstanding Scholars Fund, Guangxi Natural Science Foundation of China under grant No. 2015GXNSFDA139038, 2016GXNSFAA380054, Innovation Project of GUET Graduate Education (2017 YJCX51,2016YJCX12), Guangxi Key Laboratory of Trusted Software Focus Fund, Program for Innovative Research Team of Guilin University of Electronic Technology.

References

1. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009)
2. Allan, C., Avgustinov, P., Christensen, A.S., et al.: Adding trace matching with free variables to AspectJ. ACM SIGPLAN Not. **40**(10), 345–364 (2005)
3. Avgustinov, P., Tibble, J., Moor, O.D.: Making trace monitors feasible. ACM SIGPLAN Not. **42**(10), 589–608 (2007)

4. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. ACM SIGPLAN Not. **40**(10), 365–383 (2005)
5. Chen, F., Roşu, G.: Java-MOP: a monitoring oriented programming environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_36
6. Chen, F., D'Amorim, M., Roşu, G.: Checking and correcting behaviors of Java programs at runtime with Java-MOP. Electron. Notes Theor. Comput. Sci. **144**(4), 3–20 (2006)
7. Daian, P., et al.: RV-android: efficient parametric android runtime verification, a brief tutorial. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 342–357. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_24
8. Yamagata, Y., et al.: Runtime monitoring for concurrent systems. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 386–403. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_24
9. Giannakopoulou, D., Havelund, K.: Runtime analysis of linear temporal logic specifications. In: IEEE International Conference on Automated Software Engineering 2001 (2001)
10. Jin, D., Griffith, D., Chen, F.: An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Transf. **14**(3), 249–289 (2012)
11. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_6
12. Tischner, D.: Minimization of Büchi Automata using Fair Simulation (2016)
13. Pnueli, A.: The Temporal Logic of Programs. Weizmann Science Press of Israel, pp. 46–57 (1977)
14. Chen, F., D'Amorim, M., Roşu, G.: A formal monitoring-based framework for software development and analysis. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 357–372. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30482-1_31
15. Schwoon, S., Esparza, J.: A note on On-the-Fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_12
16. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Büchi automaton. SIAM J. Comput. **34**(5), 1159–1175 (2005)
17. Blackburn, S.M., Garner, R., Hoffmann, C., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. ACM SIGPLAN Not. **41**(10), 169–190 (2006)

Developing A New Language to Construct Algebraic Hierarchies for Event-B

James Snook(✉), Michael Butler, and Thai Son Hoang

ECS, University of Southampton, Southampton, UK
{jhs1m15,mjb,t.s.hoang}@ecs.soton.ac.uk

Abstract. This paper proposes a new extension to the Event-B modelling method to facilitate the building of hierarchical mathematical libraries to ease the formal modelling of many systems. The challenges are to facilitate building mathematical theories, be compatible with the current method and tools, and to be extensible by users within the Rodin Platform supporting Event-B.

Our contribution is a new language, called $B^\sharp$, which includes the additional features of type classes and sub-typing. The $B^\sharp$ language compiles to the current language used by the Rodin's Theory Plug-in, which ensures consistency, and also gives compatibility with the current Rodin tools. We demonstrate the advantages of the new language by comparative examples with the existing Theory Plug-in language.

Keywords: Formal methods · Event-B · Theorem prover
Mathematical extensions

1 Introduction

The Event-B method [1] and its supporting Rodin [2] are designed specifically for system modelling. Event-B incorporates mechanisms such as refinement and decomposition to cope with the system complexity. Rodin includes facilities such as animation, model checking, and theorem proving for validating and verifying the Event-B models. Often, during system development, in order to ensure system dependability, developers need to model the system's operating environment. Having extensive mathematical libraries makes this modelling task faster and easier. Building these libraries of mathematical definitions and theorems is made considerably easier with the right tools and language features.

The challenges addressed in this paper are designing a language with the required features to enable the development of consistent mathematical theories, in such a way that minimises repeated proofs. Moreover, the mathematical theories can be used within the Event-B models.

Our contribution therefore is the design of a new language, called $B^\sharp$, with features to aid the construction of mathematical libraries. The new features are designed to reason about abstract and concrete mathematical types. The $B^\sharp$ language is mapped to the current Event-B syntax as supported by Rodin and its Theory Plug-in. The mapping phase will generate necessary theorems that required to be discharged by the developers. Other proof rules will be generated which would (normally) have required a manual proof in Event-B, however, the new language will generate the proof itself.

Structure. The rest of this paper is structured as follows. Section 2 examines the mathematics that we want to model, and the relationships between mathematical types. We also summarise the modelling task in Event-B and its Theory extension. Section 3 shows the problems with the current tools by describing a case study, and presents elements of the B$^\sharp$ language to facilitate the construction of mathematical definitions and theorems. Section 4 discusses the related work and gives some conclusion of our work.

2 Background

2.1 Mathematical Data Structures

Within mathematics, the study of abstract algebra deals with abstract structures. Rather than dealing with specific functions and sets, they generalise to deal with all sets and functions that have given properties. For instance, the abstract structure of a monoid is a set S and a binary function f and an identity element e, such that:

$$\forall x, y, z \in S \cdot f(x, f(y, z)) = f(f(x, y), z), \text{ and} \tag{1}$$

$$\forall x \in S \cdot f(x, e) = x \wedge f(e, x) = x. \tag{2}$$

Property (1) declares that the function f is associative. Property (2) defines e as an identity element.

Many concrete structures are examples of this abstract type such as, addition and zero on the real numbers, or matrix multiplication with the identity matrix. Theorems and functions on the abstract type apply to all of the concrete examples, so the proof only has to be done on the abstract type.

Axioms can be added to abstract types to form new types e.g., a group is a monoid where all elements have an inverse. A group can utilise all the results from a monoid (as the group has all of the monoid's axioms), and can have new theorems provable with the new axioms.

Reasoning like this reduces the proof burden as proofs done on the abstract type do not need to be repeated by either concrete instances, or abstract types that extend the current type. The proofs are inherited by the new types.

2.2 Event-B and the Theory Plug-In

The Event-B modelling method [1] allows the modelling of discrete event systems. The Event-B modelling language is supported by the Rodin Platform [2], an open and extensible toolset for constructing formal models. To increase the ability of Event-B to model systems the Theory Plug-in [5] was added to

Rodin Platform (Rodin) allowing the extension of the Event-B mathematical language with user-defined operators and proof rules. The Theory Plug-in can be used as a theorem prover to create domain-specific mathematical theories. In [4] a 3-D Euclidean space was modelled, and used to formally verify the safety of a set of paths of Unmanned Aerial Vehicles (UAVs). The Euclidean space model would be useful to any other system requiring a safe distance is maintained. Other mathematical structures that model environment are also widely reusable e.g., two's complement arithmetic would be useful to many software system models. The aim of creating the new language B$^\sharp$ is to improve the tools for building these mathematical models for Event-B.

3 A Language for Mathematical Libraries in Event-B

3.1 A Case Study Using Theory Plug-In

This section summarises the result of a case study evaluating the ability of the current Theory Plug-in to build and use algebraic hierarchies. The case study used the Theory Plug-in to construct abstract and concrete mathematical classes, and then see how they can be related.

On the one hand, our case study shows that abstract mathematical types were representable within in the Event-B syntax. It also found that abstract types could be extended to make new abstract types, and that concrete types could be related to the abstract types. On the other hand, the following issues with the representation were identified:

1. Event-B operators are not first class members of the language, resulting in the need to encapsulate them within total functions to relate concrete types to abstract types, e.g., showing addition and zero form a monoid required a theorem such as: $zero \mapsto (\lambda x, y | x\ add\ y) \in Monoid(pNat)$ (the operator is encapsulated within a lambda construct).
2. Demonstrating that a concrete object forms an algebraic type does not make it inherit the theorems/proof rules of the algebraic type. These have to be re-written and proved (although the proof can be constructed by instantiating the theorems/proof rules of the algebraic type).
3. When making theorems about an abstract type the type required construction from its constituent parts, resulting in verbose theorem definitions. Alternatively the abstract types can be passed in and deconstructed with the Event-B projection operators making the theorem difficult to read/understand (this can be helped by the user making operators to deconstruct the abstract types).
4. The Event-B language is not able to reason about subsets as types, this resulted in the user having to manually do many well-definedness proofs.
5. Predicates in Event-B are not expressions. This makes it difficult to reason about relations (instead the *BOOL* type was used and turned back into a predicate where necessary using equality).
6. Abstract types definitions and declarations rapidly increased in complexity.

3.2 The B$^\sharp$ Language

This section gives a brief introduction to the B$^\sharp$ language, in particular focusing on the **Class** declaration, allowing the user to create new type classes, and new subtypes, fully supported by the language. A type class allows the definition of a subtype of some existing type structure such that the subtype has additional properties and operators. A type class also allows us to constrain polymorphism. For example, the following declaration defines the *ReflexRel* class:

$$\begin{aligned}&\textbf{Class}\ ReflexRel\langle T\rangle : T \times T \rightarrow Pred\\ &\quad\textbf{where}\ \forall x : T, rel : ReflexRel\langle T\rangle \cdot rel(x, x)\{\}\end{aligned} \tag{3}$$

This class declaration creates a type class *ReflexRel* a subtype of $T \times T \rightarrow Pred$, any relation in *ReflexRel* must have the additional property that all elements are related to themselves.

Some differences to the Event-B syntax can be seen immediately. The polymorphic type T can be a subtype i.e., a type created using the subtyping mechanism above. In Event-B this is the equivalent of allowing entities created with the subset syntax to be treated as types. The B$^\sharp$ language does extra work to reason about subtypes and reduce well-definedness proofs. In Event-B predicates are a different syntactic category to boolean expressions and are not first class. In B$^\sharp$, predicates are first class of type Pred.. It allows the language to create functions that return predicates without having to use the BOOL type as an intermediate.

This class declaration maps to the following underlying Event-B statement:

$$\begin{aligned}ReflexRel(t : \mathbb{P}(T)) \hat{=} \{rel | rel \in \mathbb{P}(t \times t)\\ \wedge \forall x \cdot x \in t \Rightarrow x \mapsto x \in rel\}\end{aligned} \tag{4}$$

To allow the *ReflexRel* operator to work on subtypes the Event-B power set operator $\mathbb{P}$ is used to give the type of t. *Pred* is replaced by constricting the set of rel. When using rel within an expression the mapping to Event-B will become $x \mapsto x \in rel$ when the Event-B language requires a predicate value (e.g., as in the quantifier in (4)).

The class declaration can also be used to create type classes where the type class is required to have certain elements, e.g., a monoid:

$$\begin{aligned}&\textbf{Class}\ Monoid : Setoid(ident : Monoid, op : AssocOp\langle Monoid\rangle)\\ &\quad\textbf{where}\ \forall x : Monoid \ \cdot\ op(x, ident)\ Monoid.equ\ x\\ &\quad\wedge\ \ op(ident, x)\ Monoid.equ\ x\{\}\end{aligned} \tag{5}$$

Type classes create templates for new classes. For a class to become part of the monoid type class it needs to have an identity and an associative operator that follows the rules in the **where** clauses. *Monoid* : *Setoid* means that *Monoid* is a subtype of the *Setoid* type class, which is a type which has an equivalence relation (this is created using a class declaration similar to the one above). Definition (5) maps to the following Event-B:

$$\begin{aligned}&Monoid(t : \mathbb{P}(T)) \mathrel{\hat{=}} \{setoid \mapsto ident \mapsto op \mid\\&\quad setoid \in Setoid(t) \land ident \in t \land\\&\quad op \in AssocOp(t, setoid) \land\\&\quad \forall x \cdot x \in t \Rightarrow\\&\qquad op(x \mapsto ident) \mapsto x \in Setoid\_equ(setoid) \land\\&\qquad op(ident \mapsto x) \mapsto x \in Setoid\_equ(setoid)\}\end{aligned} \tag{6}$$

$$Monoid\_Setoid(m : Monoid(\mathbb{P}(T))) \mathrel{\hat{=}} prj1(m) \tag{7}$$

$$Monoid\_ident(m : Monoid(\mathbb{P}(T))) \mathrel{\hat{=}} prj1(prj2(m)) \tag{8}$$

$$Monoid\_op(m : Monoid(\mathbb{P}(T))) \mathrel{\hat{=}} prj2(prj2(m)) \tag{9}$$

Given a instance $a = b_1 \mapsto b_2 \ldots \mapsto b_n$ $prj1(a)$ will give b_1 and $prj2(a)$ will give $b_2 \mapsto \ldots b_n$. From (6) it can be seen that the B$^\sharp$ syntax is much more concise. It is useful to see how a type becomes a member of the *Monoid* type class:

$$\textbf{Instance } Monoid(zero, add); \tag{10}$$

This will make the *pNat* type (inferred from the *zero* and *add* arguments) an instance of a *Monoid*. A proof obligation to demonstrate that addition and zero form a monoid will be generated. Proof rules, theorems and functions from the *Monoid* are re-written to rules about zero and addition and added to the *pNat* type. As these have been proved in the *Monoid* type class they do not need to be reproved.

Polymorphic types can be restricted to a given type class, e.g.:

$$\begin{aligned}&\textbf{Class } AssocOp <T : Setoid> T \times T \rightarrow T\\&\textbf{where } \forall x, y, z : T \cdot AssocOp(AssocOp(x, y), z)\ T.eq\ AssocOp(x, AssocOp(y, z))\end{aligned} \tag{11}$$

The polymorphic type T has to be a member of the *Setoid* type class (it has to have an equivalence relation). This will map to the following Event-B:

$$\begin{aligned}&AssocOp(t : \mathbb{P}(\mathbb{T}), setoid : Setoid(t)) \mathrel{\hat{=}} \{op \mid op \in t \times t \rightarrow t\\&\qquad \land \forall x, y, z \cdot x \in t \land y \in t \land z \in t\\&\qquad \Rightarrow op(op(x \mapsto y) \mapsto z) \mapsto op(x \mapsto op(y \mapsto z)) \in Setoid\_Eq(setoid)\end{aligned} \tag{12}$$

The polymorphic context in (11) ($<T : Setoid>$) becomes the arguments to the Event-B operator in (12). *Setoid_Eq* is an Event-B deconstructor created from the *Setoid* type class mapping, in the Event-B mapping this is mapped from the B$^\sharp$ *T.eq* statement.

The syntax for a **Class** statement is:

$$\textbf{Class } \gamma\langle \tau_1 : \gamma_1, \ldots, \tau_n : \gamma_n\rangle : S_1 \ldots S_m\ (s_1 : T_1, \ldots, s_p : T_p)\ \textbf{where } e_1; \ldots; e_l; \{\} \tag{13}$$

This declaration has the following meanings:

1. γ is the name chosen for the new class, e.g., *ReflexRel* in Example (3), this maps to the Event-B operator name.
2. $\langle \tau_1 : \gamma_1, \ldots, \tau_n : \gamma_n \rangle$ is the polymorphic context. γ_i are optional, and restrict τ_i to a given type class. These map to the arguments of the Event-B operator.
3. $S_1 \ldots S_m$ are super types and $(s_1 : T_1, \ldots, s_p : T_p)$ define addition structure. In the Event-B syntax they are mapped to a statement such as $\{\gamma \mapsto s_1 \mapsto \ldots \mapsto s_p | \gamma \in S_1 \land \ldots \land \gamma \in S_m \land s_1 \in T_1' \cdots \land s_p \in T_p' | \ldots\}$. With multiple inheritance any shared supertypes remain shared, the mapping for these is more complex than shown above (it requires the supertypes to be deconstructed to their last shared ancestor). This is omitted for brevity.
4. Properties $e_1; \ldots; e_l$ are predicate expressions. These create the subtype from the supertypes. Each e_i is translated to the Event-B syntax and they constrain the set returned.

Due to the space limit, we omit other features of B$^\sharp$, such as B$^\sharp$ functions, and their mapping to Event-B, or the generation of proof rules from the B$^\sharp$ class statements (to make proving easier).

4 Related Work and Conclusion

There are many examples of similar constructs in other languages. Of particular interest is Coq [3] for which there has been an extensive library of abstract algebraic structures developed [6]. The language feature which makes building this library possible is called type classes [11] originally created for Haskell. Type classes set out a structure, which types can adopt, and inherit functions from the type class. Isabelle [10] also has a similar feature allowing abstract specifications called locales [9]. Algebraic Specification [8] languages give a formal specification to datatypes rather than describing the structure of the datatype. This abstraction means that many concrete datatypes could comply with the specification, giving a similar concept to the ones described above. For example, OBJ3 [7] has a similar concepts with parameterised modules and theories. Theories define a structure for an module, parameterised types can then be restricted to models with this structure.

The novelty in the B$^\sharp$ language is not the invention of new language features but of tailoring and applying these to the Rodin toolset, mapping the extended B$^\sharp$ features to the Event-B syntax for consistency and proof purposes. The work above demonstrates a method by which this can be achieved. These new features will allow for the development of hierarchies of generic theories and the ability to develop domain-specific specialisations of these, while avoiding the need to redo proofs over similar structures for each specialisation.

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. STTT **12**(6), 447–466 (2010)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
4. Bogdiukiewicz, C., et al.: Formal development of policing functions for intelligent systems. In: 28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, 23–26 October 2017, pp. 194–204. IEEE Computer Society (2017)
5. Butler, M., Maamria, I.: Practical theory extension in Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 67–81. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_5
6. Cruz-Filipe, L., Geuvers, H., Wiedijk, F.: C-CoRN, the constructive Coq repository at Nijmegen. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 88–103. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27818-4_7
7. Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In: Goguen, J., Malcolm, G. (eds.) Software Engineering with OBJ. Advances in Formal Methods, pp. 3–167. Springer, Boston (2000). https://doi.org/10.1007/978-1-4757-6541-0_1
8. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Inform. **10**(1), 27–52 (1978)
9. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales a sectioning concept for isabelle. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 149–165. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48256-3_11
10. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
11. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 60–76. ACM (1989)

Towards the Existential Control of Boolean Networks: A Preliminary Report

Soumya Paul[1](✉), Jun Pang[1,2], and Cui Su[1]

[1] Interdisciplinary Centre for Security, Reliability and Trust, Esch-sur-Alzette, Luxembourg

[2] Faculty of Science, Technology and Communication, University of Luxembourg, Esch-sur-Alzette, Luxembourg

{soumya.paul,jun.pang,cui.su}@uni.lu

Abstract. Given a Boolean network BN and a subset $\mathcal{A}$ of attractors of BN, we study the problem of identifying a minimal subset C_{BN} of vertices of BN, such that the dynamics of BN can reach from a state $\mathbf{s}$ in any attractor $A_s \in \mathcal{A}$ to any attractor $A_t \in \mathcal{A}$ by controlling (toggling) a subset of vertices in C_{BN} in a single time step. We describe a method based on the decomposition of the network structure into strongly connected components called 'blocks'. The control subset can be locally computed for each such block and the results then merged to derive the global control subset C_{BN}. This potentially improves the efficiency for many real-life networks that are large but modular and well-structured. We are currently in the process of implementing our method in software.

1 Introduction

Systems biology, with the help of mathematical modelling, has revolutionised the human diseasome research and paved the way towards the development of new therapeutic approaches and personalised medicine. Such therapies target specific proteins within the cellular systems aiming to drive it from a 'diseased' state to a 'healthy' state. However, it has been observed that disease-networks are intrinsically robust against perturbations due to the inherent diversity and redundancy of compensatory signalling pathways [2]. This greatly reduces the efficacy of single-target drugs. Hence, rather than trying to design selective ligands that target individual receptors only, network polypharmacology seeks to modify multiple cellular targets to tackle the compensatory mechanisms and robustness of disease-associated cellular systems. This motivates the question of identifying multiple drug targets using which the network can be 'fully controlled', i.e. driven from any (diseased) state to any desired target (healthy) state. Furthermore, for the feasibility of the synthesis of such drugs, the number of such targets should be minimised. However, biological networks are intrinsically large (number of components, parameters, interactions, etc.) which results

https://doi.org/10.1007/978-3-319-99933-3_10

in an exponentially increasing number of potential drug target combination making a purely experimental approach quickly infeasible. This reinforces the need of mathematical modelling and computational techniques.

Boolean networks (BNs), first introduced by Kauffman [5], is a popular and well-established framework for modelling gene regulatory networks (GRNs) and their associated signalling pathways. Its main advantage is that it is simple and yet able to capture the important dynamical properties of the system under study, thus facilitating the modelling of large biological systems as a whole. The states of a BN are tuples of 0s and 1s where each element of the tuple represents the level of activity of a particular protein in the GRN or the signalling pathway it models - 0 for inactive and 1 for active. The BN is assumed to evolve dynamically by moving from one state to the next governed by a Boolean function for each of its components. The steady state behaviour of a BN is given by its subset of states called *attractors* to one of which the dynamics eventually settles down. In biological context, attractors are hypothesised to characterise cellular phenotypes [5] and also correspond to functional cellular states such as proliferation, apoptosis, differentiation, etc. [3]. The *control* of a BN therefore refers to the reprogramming/changing of the parameters of the BN (functions, values of variables, etc.) so that its dynamics eventually reaches a desired attractor or steady state.

The full control of linear networks is a well-studied problem [4] and such control strategies have been proposed over the years. Recent work on network controllability has shown that full controllability and reprogramming of intercellular networks can be achieved by a minimum number of control targets [7]. However, the full control of non-linear networks is apparently more challenging predominantly due to the explosion of the potential search space with the increase in the network size. There has not been a lot of work in this regard. Kim et al. [6] developed a method to identify the so-called 'control kernel' which is a minimal set of nodes for fully controlling a biological network. But, their method is based on the construction of the full state transition graph of the network and as such does not scale well for large networks.

In many cases, only some of the attractors of the BNs are 'biologically relevant', i.e. correspond to meaningful expressions of the modelled GRNs. Thus, focussing on only the relevant attractors might help reduce the complexity of the control problem while still being biologically meaningful.

Our Contributions. In this work, we report the initial results on a method for the control of Boolean networks that exploits both their structural and dynamic properties, as shown inevitable in [1]. More precisely, given a Boolean network BN and a set of 'relevant' attractors $\mathcal{A}$ of BN, the method computes a minimal set of variables (the *minimal control set*), such that starting from an initial attractor $A_s \in \mathcal{A}$ and by controlling specific subsets of these variables in a *single time-step*, the BN can (potentially) reach any desired target attractor $A_t \in \mathcal{A}$ when left to evolve on its own according to its original dynamics. A welcome side-effect of the method is that when $\mathcal{A}$ is the set of all attractors of BN, it gives the minimal set of vertices for fully controlling BN. We use an approach that we have developed for the problem of target control (driving the BN to a given single target attractor) of BNs, based on the decomposition of its network structure into strongly connected components called 'blocks'. Although the method can be applied on the entire BN in one-go, we believe that using the decomposition-based approach can greatly increase its efficiency on large real-life biological networks whose BN models have well-behaved modular structure. This is work-in-progress and we are currently implementing our method in software to test its effectiveness on various networks.

2 Background and Notations

Let $N = \{1, 2, \ldots, n\}$ where $n \geq 1$. A *Boolean network* is a tuple $\mathsf{BN} = (\mathbf{x}, \mathbf{f})$ where $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ such that each x_i is a Boolean variable and $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ is a tuple of Boolean functions over $\mathbf{x}$. In what follows, i will always range over N, unless stated otherwise. A Boolean network $\mathsf{BN} = (\mathbf{x}, \mathbf{f})$ may be viewed as a directed graph $\mathcal{G}_{\mathsf{BN}} = (V, E)$, where $V = \{v_1, v_2 \ldots, v_n\}$ is the set of *vertices* or *nodes* (intuitively, v_i corresponds to the variable x_i for all i) and for every $i, j \in N$, there is a directed edge from v_j to v_i, often denoted as $v_j \to v_i$, if and only if f_i depends on x_j. Thus V is ordered according to the ordering of $\mathbf{x}$. For any vertex $v_i \in V$, we let $\mathsf{ind}(v_i) = i$ be the index of v_i in this ordering. For any subset W of V, $\mathsf{ind}(W) = \{\mathsf{ind}(v) | \; v \in W\}$. A *path* from a vertex v to a vertex v' is a (possibly empty) sequence of edges from v to v' in $\mathcal{G}_{\mathsf{BN}}$. For any vertex $v \in V$ we define its set of *parents* as $\mathsf{par}(v) = \{v' \in V \mid v' \to v\}$ and for any subset W of V, $\mathsf{par}(W) = \{\mathsf{par}(v) \mid v \in W\}$. For the rest of the exposition, we assume that an arbitrary but fixed network BN of n variables is given to us and $\mathcal{G}_{\mathsf{BN}} = (V, E)$ is its associated directed graph.

A *state* $\mathbf{s}$ of BN is an element in $\{0, 1\}^n$. Let $\mathbf{S}$ be the set of states of BN. For any state $\mathbf{s} = (s_1, s_2, \ldots, s_n)$, and for every i, the value of s_i, often denoted as $\mathbf{s}[i]$, represents the value that the variable x_i takes when the BN 'is in state $\mathbf{s}$'. For some i, suppose f_i depends on $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$. Then $f_i(\mathbf{s})$ will denote the value $f_i(\mathbf{s}[i_1], \mathbf{s}[i_2], \ldots, \mathbf{s}[i_k])$. For two states $\mathbf{s}, \mathbf{s}' \in \mathbf{S}$, the *Hamming distance* between $\mathbf{s}$ and $\mathbf{s}'$ will be denoted as $\mathsf{hd}(\mathbf{s}, \mathbf{s}')$ and $\arg(\mathsf{hd}(\mathbf{s}, \mathbf{s}')) \subseteq N$ will denote the set of indices in which $\mathbf{s}$ and $\mathbf{s}'$ differ. For a state $\mathbf{s}$ and a subset $\mathbf{S}' \subseteq \mathbf{S}$, the Hamming distance between $\mathbf{s}$ and $\mathbf{S}'$ is defined as $\mathsf{hd}(\mathbf{s}, \mathbf{S}') = \min_{\mathbf{s}' \in \mathbf{S}'} \mathsf{hd}(\mathbf{s}, \mathbf{s}')$. We let $\arg(\mathsf{hd}(\mathbf{s}, \mathbf{S}'))$ denote the set of subsets of N such that $I \in \arg(\mathsf{hd}(\mathbf{s}, \mathbf{S}'))$ if and only if I is a set of indices of the variables that realise $\mathsf{hd}(\mathbf{s}, \mathbf{S}')$.

We assume that the Boolean network starts initially in a state $\mathbf{s}_0$ and its state changes in every discrete time-step according to the update functions $\mathbf{f}$. In this work, we shall deal with the asynchronous updating scheme but all our results transfer to the synchronous updating scheme as well. Suppose $\mathbf{s}_0 \in \mathbf{S}$ is an initial state of BN. The *asynchronous evolution* of BN is a function $\xi : \mathbb{N} \to \wp(\mathbf{S})$ such that $\xi(0) = \mathbf{s}_0$ and for every $j \geq 0$, if $\mathbf{s} \in \xi(j)$ then $\mathbf{s}' \in \xi(j + 1)$ if and only if either $\mathsf{hd}(\mathbf{s}, \mathbf{s}') = 1$ and $\mathbf{s}'[i] = f_i(\mathbf{s})$ where $i = \arg(\mathsf{hd}(\mathbf{s}, \mathbf{s}'))$ or $\mathsf{hd}(\mathbf{s}, \mathbf{s}') = 0$ and there exists i such that $\mathbf{s}'[i] = f_i(\mathbf{s})$.

The dynamics of a Boolean network can be represented as a *state transition graph* or a *transition system (TS)*. The *transition system* of BN, denoted as $\mathsf{TS}_{\mathsf{BN}}$

is a tuple $(\mathbf{S}, \rightarrow)$ where the vertices are the set of states $\mathbf{S}$ and for any two states $\mathbf{s}$ and $\mathbf{s}'$ there is a directed edge from $\mathbf{s}$ to $\mathbf{s}'$, denoted $\mathbf{s} \rightarrow \mathbf{s}'$, if and only if either $\mathsf{hd}(\mathbf{s}, \mathbf{s}') = 1$ and $\mathbf{s}'[i] = f_i(\mathbf{s})$ where $i = \arg(\mathsf{hd}(\mathbf{s}, \mathbf{s}'))$ or $\mathsf{hd}(\mathbf{s}, \mathbf{s}') = 0$ and there exists i such that $\mathbf{s}'[i] = f_i(\mathbf{s})$.

For any state $\mathbf{s} \in \mathbf{S}$, $\mathsf{pre}_{\mathsf{TS}}(\mathbf{s}) = \{\mathbf{s}' \in \mathbf{S} \mid \mathbf{s}' \rightarrow \mathbf{s}\}$ contains all the states that can reach $\mathbf{s}$ by performing a single transition in TS. For a subset $\mathbf{S}'$ of $\mathbf{S}$, $\mathsf{pre}_{\mathsf{TS}}(\mathbf{S}') = \bigcup_{\mathbf{s} \in \mathbf{S}'} \mathsf{pre}_{\mathsf{TS}}(\mathbf{s})$. A *path* from a state $\mathbf{s}$ to a state $\mathbf{s}'$ is a (possibly empty) sequence of transitions from $\mathbf{s}$ to $\mathbf{s}'$ in $\mathsf{TS}_{\mathsf{BN}}$. A path from a state $\mathbf{s}$ to a subset $\mathbf{S}'$ of $\mathbf{S}$ is a path from $\mathbf{s}$ to any state $\mathbf{s}' \in \mathbf{S}'$. For a state $\mathbf{s} \in \mathbf{S}$, $\mathsf{reach}_{\mathsf{TS}}(\mathbf{s})$ denotes the set of states $\mathbf{s}'$ such that there is a path from $\mathbf{s}$ to $\mathbf{s}'$ in TS.

An *attractor* A of $\mathsf{TS}_{\mathsf{BN}}$ (or of BN) is a subset of states of $\mathbf{S}$ such that for every $\mathbf{s} \in A$, $\mathsf{reach}_{\mathsf{TS}_{\mathsf{BN}}}(\mathbf{s}) = A$. Any state which is not part of an attractor is a *transient state*. An attractor A of BN is said to be reachable from a state $\mathbf{s}$ if $\mathsf{reach}_{\mathsf{TS}_{\mathsf{BN}}}(\mathbf{s}) \cap A \neq \emptyset$. Attractors represent the stable behaviour of the BN according to the dynamics. For an attractor A of BN, the *weak basin* or simply the *basin* of attraction of A, denoted $\mathsf{bas}_{\mathsf{TS}_{\mathsf{BN}}}(A)$, is a subset of states of $\mathbf{S}$ such that $\mathbf{s} \in \mathsf{bas}_{\mathsf{TS}_{\mathsf{BN}}}(A)$ if $\mathsf{reach}_{\mathsf{TS}_{\mathsf{BN}}}(\mathbf{s}) \cap A \neq \emptyset$. A *control* C is a (possibly empty) subset of N. For a state $\mathbf{s} \in \mathbf{S}$, the *application of control* C to $\mathbf{s}$, denoted $\mathsf{C}(\mathbf{s})$ is defined as the state $\mathbf{s}' \in \mathbf{S}$ such that $\mathbf{s}'[i] = (1 - \mathbf{s}[i])$ if $i \in \mathsf{C}$ and $\mathbf{s}'[i] = \mathbf{s}[i]$, otherwise. Henceforth, we drop the subscripts TS or BN or both when no ambiguity arises.

Control Problems: In this work we shall exclusively deal with the notion of *existential control* in that, after the control C is applied to a state $\mathbf{s}$, there 'exists' a path from $\mathsf{C}(\mathbf{s})$ to the desired target attractor and also perhaps to other non-target attractors. This is different from the notion of *absolute control* dealt with in [10] where after the control, $\mathsf{C}(\mathbf{s})$ is 'guaranteed' to reach the target attractor. Although the techniques applied for the computation of the minimal control are similar in both cases, there are certain fundamental differences. Therefore, here we are interested in the following control problems given a network BN. Note that for us, the control is applied in a single time step (hence simultaneously) to the state $\mathbf{s}$ under consideration.

1. **Minimal existential target control:** Given a state $\mathbf{s} \in \mathbf{S}$ and a 'target attractor' A_t of BN, it is a control $\mathsf{C}_{\mathbf{s} \rightarrow A_t}$ such that after the application of $\mathsf{C}_{\mathbf{s} \rightarrow A_t}(\mathbf{s})$, BN can eventually reach A_t and $\mathsf{C}_{\mathbf{s} \rightarrow A_t}$ is a minimal such subset.
2. **Minimal existential all-pairs control:** Given a set $\mathcal{A} = \{A_1, A_2, \ldots, A_p\}$, $p \geq 2$, of attractors of BN, it is a minimal subset $\mathsf{C}_{\mathcal{A}}$ of N such that for any pair $A_i, A_j \in \mathcal{A}$ of attractors, there is a state $\mathbf{s} \in A_i$, such that $\mathsf{C}_{\mathbf{s} \rightarrow A_j} \subseteq \mathsf{C}_{\mathcal{A}}$.
3. **Minimal existential full control:** C_{BN} is the minimal existential all-pairs control $\mathsf{C}_{\mathcal{A}}$ when $\mathcal{A}$ is the set of all attractors of BN.

In this work we shall use ideas from the decomposition-based approach of [10] to compute (2) and (3). We first give the relevant definitions and results.

Let SCC denote the set of maximal strongly connected components (SCCs) of $\mathcal{G}_{\mathsf{BN}}$. A *basic block* B is a subset of nodes of BN such that $B = (S \cup \mathsf{par}(S))$ where S is a maximal SCC of $\mathcal{G}_{\mathsf{BN}}$. Let $\mathcal{B}$ denote the set of basic blocks of BN. The union of two or more basic blocks will also be called a *block*. Using the set of basic blocks as vertices, we can form a directed graph $\mathcal{G}_{\mathcal{B}} = (\mathcal{B}, E_{\mathcal{B}})$, called the *block graph* of BN, where for any pair of basic blocks $B', B \in \mathcal{B}, B' \neq B$, there is a directed edge from B' to B if and only if $B' \cap B \neq \emptyset$ and for every $v \in B' \cap B$, $\mathsf{par}(v) \cap B = \emptyset$. In such a case, B' is called a *parent* block of B and v is called a *control node* for B. The set of parent blocks of B is denoted as $\mathsf{par}(B)$.

A block is called *elementary* if $\mathsf{par}(B) = \emptyset$ and *non-elementary* otherwise. We shall henceforth assume that BN has k basic blocks, $|\mathcal{B}| = k$, and $\mathcal{G}_{\mathsf{BN}}$ is topologically sorted as $\{B_1, B_2, \ldots, B_k\}$. Given how $\mathcal{G}_{\mathsf{BN}}$ is constructed, it will be a directed acyclic graph and hence can always be topologically sorted. Note that for every $j : 1 \leq j \leq k$, $(\bigcup_{\ell=1}^{j} B_\ell)$ is an elementary block. We shall denote it as $\overline{B}_j$ and let $B_j^- = (B_j \setminus \overline{B}_{j-1})$. For two basic blocks B and B' where B is non-elementary, B' is said to be an *ancestor* of B if there is a path from B' to B in the block graph $\mathcal{G}_{\mathcal{B}}$. The *ancestor-closure* of a basic block B, denoted $\mathsf{ac}(B)$ is defined as the union of B with all its ancestors. Note that $\mathsf{ac}(B)$ is an elementary block and so is $(\mathsf{ac}(B) \setminus B^-)$, denoted as $\mathsf{ac}(B)^-$.

For a block B of BN, its state space is $\{0,1\}^{|B|}$ and is denoted as $\mathbf{S}_B$. For any state $\mathbf{s} \in \mathbf{S}$, where $\mathbf{s} = (s_1, s_2, \ldots, s_n)$, the projection of $\mathbf{s}$ to B, denoted $\mathbf{s}|_B$ is the tuple obtained from $\mathbf{s}$ by suppressing the values of the variables not in B. Let B_1 and B_2 be two blocks of BN and let $\mathbf{s}_1$ and $\mathbf{s}_2$ be states of B_1 and B_2, respectively. $\mathbf{s}_1 \otimes \mathbf{s}_2$ is defined (called *crossable*) if there exists a state $\mathbf{s} \in \mathbf{S}_{B_1 \cup B_2}$ such that $\mathbf{s}|_{B_1} = \mathbf{s}_1$ and $\mathbf{s}|_{B_2} = \mathbf{s}_2$. $\mathbf{s}_1 \otimes \mathbf{s}_2$ is then defined to be this unique state $\mathbf{s}$. For any subsets $\mathbf{S}_1$ and $\mathbf{S}_2$ of $\mathbf{S}_{B_1}$ and $\mathbf{S}_{B_2}$ resp. $\mathbf{S}_1 \otimes \mathbf{S}_2$ is a subset of $\mathbf{S}_{B_1 \cup B_2}$ and is defined as: $\mathbf{S}_1 \otimes \mathbf{S}_2 = \{\mathbf{s}_1 \otimes \mathbf{s}_2 \mid \mathbf{s}_1 \in \mathbf{S}_1, \mathbf{s}_2 \in \mathbf{S}_2 \text{ and } \mathbf{s}_1, \mathbf{s}_2 \text{ are crossable}\}$. The cross operation can be defined for more than two states $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_k$, as $\mathbf{s}_1 \otimes \mathbf{s}_2 \otimes \ldots \mathbf{s}_k = (((\mathbf{s}_1 \otimes \mathbf{s}_2) \otimes \ldots) \otimes \mathbf{s}_k)$. The cross operation can be similarly lifted to more than two sets of states.

The TS TS_B of an elementary block B of BN is defined similarly to the TS of BN, which can indeed be done as the update functions do not depend on vertices outside B. The attractors, basin of attractions, etc. of such a TS is also defined similarly. The TSs of a non-elementary basic block B are 'realised' by the basins of attractions of the attractors of $\mathsf{ac}(B)^-$, each such attractor realising a different TS. Thus, if A is an attractor of $\mathsf{ac}(B)^-$ then TS_B realised by $\mathsf{bas}(A)$ has set of states $\mathbf{S}$ which the maximum subset of $\mathbf{S}_{\mathsf{ac}(B)}$ such that $\mathbf{S}|_{\mathsf{ac}(B)^-} = \mathsf{bas}(A)$. The transitions are then defined as usual. The following is a key result, a counterpart of which was proved in [10], saying that the 'global' attractors of BN and their basins can be computed by first computing the 'local' attractors and basins of the basic blocks and then merging them using the cross operation.

Theorem 1 **([10]).** *A is an attractor of BN iff there exist attractors A_j of B_j where $A_j = A|_{B_j}$ for all $j : 1 \leq j \leq k$ and $A = \otimes_j A_j$. Furthermore, $A|_{\mathsf{ac}(B_j)}$ is an attractor of $\mathsf{ac}(B_j)$ and $\mathsf{bas}(A) = \otimes_j \mathsf{bas}(A_j)$ w.r.t. their TSs.*

3 Results

In this section we develop our method for solving control problem (2). We first describe a 'global' approach that works on the entire BN and then modify it to exploit the decomposition-based approach of [10]. For simplicity, we assume that every attractor of BN is a single state with a self loop. The methods can be generalised for the case where an attractor can comprise of two or more states.

First, note that given a state $\mathbf{s}$ and an attractor A, for BN to potentially end up in A after the application of a control C, it is necessary and sufficient that there is a path from $\mathsf{C}(\mathbf{s})$ to A in $\mathsf{TS}_{\mathsf{BN}}$ which means, by definition, that $\mathsf{C}(\mathbf{s}) \in \mathsf{bas}(A)$. Thus given a set $\mathcal{A}$ of attractors of BN to compute $\mathsf{C}_{\mathcal{A}}$ it is enough to compute the basins of the attractors in $\mathcal{A}$. This can be done by a repeated application of the $\mathsf{pre}(\cdot)$ operator on an attractor till a fixed point is reached. See [9] for a detailed description of this fixed point procedure.

So, assume that the given set of attractors $\mathcal{A}$ is sorted as $\{A_1, A_2, \ldots, A_p\}$. We then construct a $p \times p$ matrix M whose entries are subsets of N and are defined as: for every $I \subseteq N$, $I \in \mathsf{M}_{ij}$ if and only if $I = \arg(\mathsf{hd}(\mathbf{s}, \mathbf{s}'))$ where $\mathbf{s} \in A_i$ and $\mathbf{s}' \in \mathsf{bas}(A_j)$. That is, for every pair of attractors A_i and A_j the entries of M_{ij} record the indices of the variables that need to be toggled in state $\mathbf{s} \in A_i$ to end up in any of the states of the basin of A_j. The minimal all-pairs control $\mathsf{C}_{\mathcal{A}}$ is then nothing but a minimal subset of N such that for every i, j there exists $I \in \mathsf{M}_{ij}$ such that $I \subseteq \mathsf{C}_{\mathcal{A}}$.

We now describe a method to compute the set $\mathsf{C}_{\mathcal{A}}$ based on the power-set lattice of N, denoted by $\mathcal{L}$. Let $\ell : \mathcal{L} \rightarrow \wp(N \times N)$ be a labelling function that labels the elements of $\mathcal{L}$ with tuples in $(N \times N)$ defined as follows. For any element L of $\mathcal{L}$, $(i,j) \in \ell(L)$ iff $L \in \mathsf{M}_{ij}$. Let ℓ^* denote the closure of the labelling function of $\mathcal{L}$ under subsets, defined as: for every element L of $\mathcal{L}$, $\ell^*(L) = \bigcup_{L' \subseteq L} \ell(L')$. Finally, $\mathsf{C}_{\mathcal{A}}$ is any minimal element L of $\mathcal{L}$ such that $\ell^*(L) = (\{1,2,\ldots,p\} \times \{1,2,\ldots,p\}) \setminus \{(i,i) \mid i \in \{1,2,\ldots,p\}\}$. Control problem (3) is a special case of (2) where $\mathcal{A}$ is the set of all attractors of BN. For solving (3), given a BN as input, we can first apply any of the methods available in the literature (e.g., see [8]) to compute the set of all attractors $\mathcal{A}$ of BN, and then invoke the above method.

In general, the problem of computing $\mathsf{C}_{\mathcal{A}}$ given the matrix M is NP-hard. Moreover, given a BN and an attractor A as input, the problem of computation of the strong basin of A is PSPACE hard. Hence, the control problem (2) is at least PSPACE-hard and so unlikely to have efficient algorithms for the general case. However, in [10] we show that using a decomposition-based approach we can improve the efficiency for many modular well-structured networks. We now describe a similar approach for solving control problem (2) [and hence (3)].

The method is iterative where instead of computing the basin of attractions of the given attractors for the entire BN in one-go, we decompose the BN into blocks, as described in the previous section, and compute the basins and also the minimal control w.r.t the basins of each such block. The basin of an attractor in a block can once again be computed using a repeated application of the $\mathsf{pre}(\cdot)$ operator in that block. The details are given in [9].

Suppose we are given a BN and a set of attractors $\mathcal{A}$ sorted as $\{A_1, A_2, \ldots, A_p\}$ as input. We proceed in the following steps:

1. We decompose BN into basic blocks $\mathcal{B}$, form the block graph $\mathcal{G}_{\mathcal{B}}$ and topologically sort it to obtain an ordering of the blocks as $\mathcal{B} = \{B_1, B_2, \ldots, B_k\}$.
2. Proceeding in the sorted order, for each block B_j we repeat the steps below:
   (a) Let $\hat{B}_j = (B_j \setminus (\bigcup_{r<j} B_r))$ and $I_j = \mathsf{ind}(\hat{B}_j)$.
   (b) Let M^j be a $p \times p$ matrix whose entries are subsets of I_j.
   (c) Note that by Theorem 1, $A_r|_{\mathsf{ac}(B_j)}$ is an attractor of B_j, for every $r : 1 \leq r \leq p$. For every r, we compute $\mathsf{bas}(A_r|_{\mathsf{ac}(B_j)})$.
   (d) We populate the matrix M^j as: for every $I \subseteq I_j$, $I \in \mathsf{M}^j_{qr}$ if and only if $I = (\arg(\mathsf{hd}(\mathbf{s}|_{\hat{B}_j}, \mathbf{s}'|_{\hat{B}_j})))$ for some $\mathbf{s} \in A_q|_{\mathsf{ac}(B_j)}$ and $\mathbf{s}' \in \mathsf{bas}(A_r|_{\mathsf{ac}(B_j)})$.
   (e) Let $\mathcal{L}_j$ be the subset lattice of I_j and let ℓ_j label the elements of $\mathcal{L}_j$ with tuples in $(I_j \times I_j)$ such that for $L \in \mathcal{L}_j$, $(q, r) \in \ell_j(L)$ iff $L \in \mathsf{M}^j_{qr}$.
   (f) Let ℓ^*_j denote the closure of ℓ_j under subsets and let $\mathsf{C}^j_{\mathcal{A}}$ be any minimal element L of $\mathcal{L}_j$ such that $\ell^*(L) = ((\{1, 2, \ldots, p\} \times \{1, 2, \ldots, p\}) \setminus \{(i, i) \mid i \in \{1, 2, \ldots, p\}\})$
3. Finally we let $\mathsf{C}_{\mathcal{A}} = \bigcup_{j=1}^{k} \mathsf{C}^j_{\mathcal{A}}$.

The above approach is worked-out in details on a toy example in [9].

4 Conclusion

In this report, we describe work-in-progress on the development of a procedure for the computation of a minimal subset of nodes required for the existential control of a given BN. Our procedure can be applied on the entire BN in one-go or on the 'blocks' of the BN locally and then later combined to derive the global control, whereby taking advantage of the decomposition-based approach towards the problem of target control of BNs that we have developed in [10]. We are currently implementing our procedure in software to test its efficacy and efficiency on various real-life and random BNs. We believe that our decomposition-based approach has great potential to efficiently solve the control problem for large real-life biological networks modelled as BNs that are modular and well-structured.

References

1. Gates, A.J., Rocha, L.M.: Control of complex networks requires both structure and dynamics. Sci. Rep. **6**, 24456 (2016)
2. Hopkins, A.L.: Network pharmacology: the next paradigm in drug discovery. Nat. Chem. Biol. **4**, 682–690 (2008)
3. Huang, S.: Genomics, complexity and drug discovery: insights from Boolean network models of cellular regulation. Pharmacogenomics **2**(3), 203–222 (2001)
4. Kalman, R.E.: Mathematical description of linear dynamical systems. J. Soc. Ind. Appl. Math. **1**(2), 152–192 (1963)
5. Kauffman, S.: Homeostasis and differentiation in random genetic control networks. Nature **224**, 177–178 (1969)
6. Kim, J., Park, S.M., Cho, K.H.: Discovery of a kernel for controlling biomolecular regulatory networks. Sci. Rep. **3**, 2223 (2013)

7. Liu, Y.Y., Slotine, J.J., Barabási, A.L.: Controllability of complex networks. Nature **473**, 167–173 (2011)
8. Mizera, A., Pang, J., Qu, H., Yuan, Q.: Taming asynchrony for attractor detection in large Boolean networks. In: IEEE/ACM TCBB (2018, in press)
9. Paul, S., Pang, J., Su, C.: Towards the existential control of boolean networks (extended abstract). Technical report, UL (2018). http://arxiv.org/abs/1806.10927
10. Paul, S., Su, C., Pang, J., Mizera, A.: A decomposition-based approach towards the control of Boolean networks. In: Proceedings of ACM-BCB 2018. ACM (2018, in press)

Timing and Scheduling

Statistical Model Checking of Response Times for Different System Deployments

Bernhard K. Aichernig[1], Severin Kann[2], and Richard Schumi[1](✉)

[1] Institute of Software Technology, Graz University of Technology, Graz, Austria
{aichernig,rschumi}@ist.tugraz.at
[2] AVL List GmbH, Graz, Austria
Severin.Kann@avl.com

Abstract. Performance testing is becoming increasingly important for interactive systems. Evaluating their performance with respect to user expectations is complex, especially for different system deployments. Various load-testing approaches and performance-simulation methods aim at such analyses. However, these techniques have certain disadvantages, like a high testing effort for load testing, and a questionable model accuracy for simulation methods. Hence, we propose a combination of both techniques. We apply statistical model checking with a learned timed model and evaluate the results on the real system with hypothesis testing. Moreover, we check the established hypotheses of a reference system on various system deployments (configurations), like different hardware or network settings, and analyse the influence on the performance. Our method is realised with a property-based testing tool that is extended with algorithms from statistical model checking. We illustrate the feasibility of our technique with an industrial case study of a web application.

Keywords: Statistical model checking · Model-based testing
System deployments · Property-based testing · Performance
Response time · Stochastic user profiles · Web-service application

1 Introduction

Analysing the performance of a system for specific usage scenarios is a difficult task. It becomes even more cumbersome, when a system is deployed to customers and should still provide certain performance properties for different hardware or network settings. We propose a performance evaluation method that applies statistical model checking (SMC) on a timed model from a reference system in order to derive hypotheses that allow us to verify system deployments. SMC [1] is an evaluation method that answers qualitative or quantitative questions, which are expressed as properties of a stochastic model or system. In contrast to existing load-testing approaches, we can perform a fast evaluation with a model, and in contrast to model-based methods, we verify the results of the model evaluation on real systems.

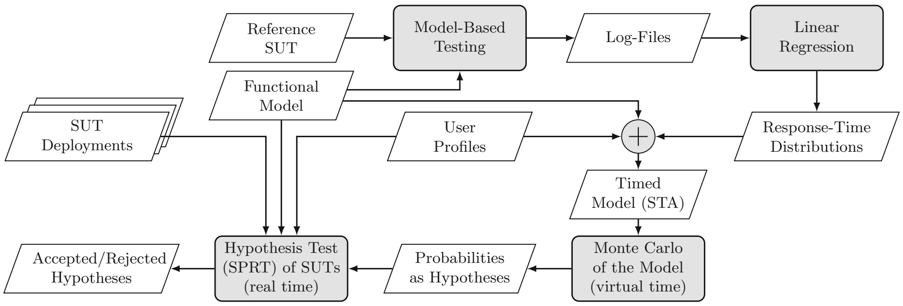

Reference SUT
Model-Based Testing
Log-Files
Linear Regression
Functional Model
SUT Deployments
User Profiles
Response-Time Distributions
Timed Model (STA)
Accepted/Rejected Hypotheses
Hypothesis Test (SPRT) of SUTs (real time)
Probabilities as Hypotheses
Monte Carlo of the Model (virtual time)

Fig. 1. Overview of the data flow of our method.

Our approach is realised with a property-based testing (PBT) tool that was extended with SMC algorithms [5]. PBT is a random testing method that tries to falsify properties that define the expected behaviour of a system-under-test (SUT). PBT tools generate random inputs and check if a property holds.

Previously, we have demonstrated how learned models can be applied to estimate the probability that a system can satisfy certain response-time thresholds, and we verified the resulting probability with hypothesis testing on the SUT. Now, we derive hypotheses about the response time from a reference SUT, in order to check, if deployments of this SUT with a different hardware/network setup have a similar performance, i.e. they satisfy the same hypotheses.

The process of our method is illustrated in Fig. 1. (1) We perform model-based testing with a functional model and capture the response times of requests of a reference SUT as log-data. We run multiple testing processes concurrently in order to obtain response times for simultaneous requests. (2) The log-files are then taken as input for a linear regression, which gives us response-time distributions. (3) These distributions and stochastic user profiles are integrated into the functional model, resulting in a combined stochastic timed automata (STA) [8] model. (4) Next, we perform a Monte Carlo simulation of this model in order to obtain answers for the question: "What is the probability that the response time of each user within a user population is under a certain threshold?". (5) The resulting probabilities serve us as hypotheses in order to check if deployments of the SUT can satisfy the same response-time thresholds as the reference system. We test the deployments with the sequential probability ratio test (SPRT) [38], a form of hypothesis testing that can usually be performed with fewer samples than a Monte Carlo simulation.

Related Work. A number of related approaches in the area of PBT are concerned with testing concurrent software [16,22,31]. The closest related work we found in this area was from Arts [7]. It shows a load-testing approach with QuickCheck that can run user scenarios on an SUT in order to determine the maximum supported number of users. In contrast to our approach, Arts does not consider stochastic user profiles and the user scenarios are only tested on an SUT, but not simulated at model-level.

Related work is also in the area of load testing [9,29]. For example, Draheim et al. [17] demonstrated a load-testing approach that simulates realistic user behaviour with stochastic models. Moreover, a number of related tools, like Neoload perform load testing with user populations [35]. In contrast to our work, load testing is mostly performed directly on an SUT. With our approach, we want to simulate user populations on the model-level as well. There are also many approaches that focus only on a simulation on the model-level [10,12,14,27,32,40], but with our method we can also directly test an SUT within the same tool.

Another domain with related work, is deployment testing. For example, various related approaches apply a performance analysis of system deployments [28,36,39]. However, in contrast to our work, they do not apply a model that is derived from a reference SUT in order to evaluate the performance of SUT deployments under specific usage scenarios.

The most related tool is UPPAAL SMC [13]. Similar to our approach, it provides SMC of priced timed automata, which can simulate user populations. It also supports testing implementations, but for this a test adapter is required, which, e.g., handles the form-data creation. In contrast, we can use PBT features, like data generators in order to automatically generate form data, and we can model in a programming language. This helps testers, who are already familiar with this language, since they do not have to learn new notations.

To the best of our knowledge our work is novel: no other work performs SMC on a learned timed model of a reference SUT to derive hypotheses that are verified on SUT deployments in order to check, if they provide comparable response times for given user profiles.

Contribution. This paper builds upon our previous work [37], where we introduced our model-based prediction method that enables an efficient test of the predictions on the SUT. We evaluated this approach on an industrial web-service application. However, it was only tested on one reference system without any deployments. Hence, this work presents the following novel contributions. (1) The major contribution is the new application of our method to analyse the performance of system deployments applying hypotheses that were derived from a reference SUT. This allows software companies to give their customers recommendations for the hardware/network requirements of a system that should satisfy certain performance properties. (2) Another contribution is the additional evaluation of our method by applying it to several deployments of an industrial web-service application. This helps to find possible limitations of our approach and emphasizes its generality.

Structure. First, Sect. 2 introduces the background of SMC, PBT and stochastic timed automata based on previous work [37]. Then, in Sect. 3 we illustrate our method with an example. In Sect. 4, we evaluate our approach with an industrial web-service application as reference SUT, and we check multiple deployments with other hardware/network configurations. Finally, we conclude in Sect. 5.

2 Background

2.1 Statistical Model Checking (SMC)

SMC is a verification method that evaluates certain properties of a stochastic model. These properties are usually defined with (temporal) logics, and they can describe quantitative and qualitative questions. For example, questions, like "What is the probability that the model satisfies a property?" or "Is the probability that the model satisfies a property above or below a certain threshold?". In order to answer such questions, a statistical model checker produces samples, i.e. random walks on the stochastic model and checks whether the property holds for these samples. Various SMC algorithms are applied, in order to compute the total number of samples needed to find an answer for a specific question, or to compute a stopping criterion. This criterion determines when we can stop sampling, because we have found an answer with a required certainty. In this work, we focus on the following algorithms, which are commonly used in the SMC literature [13, 25, 26].

Monte Carlo Simulation with Chernoff-Hoeffding Bound. The algorithm computes the required number of simulations n in order to estimate the probability γ that a stochastic model satisfies a Boolean property. The procedure is based on the Chernoff-Hoeffding bound [20] that provides a lower limit for the probability that the estimation error is below a value ϵ. Assuming a confidence $1 - \delta$, the required number of simulations n can be calculated as follows:

$$n \geq \frac{1}{2\epsilon^2} \ln\left(\frac{2}{\delta}\right)$$

The n simulations represent Bernoulli random variables $X_1, \ldots, X_n$ with outcome $x_i = 1$, if the property holds for the i-th simulation run, and $x_i = 0$ otherwise. Let the estimated probability be $\bar{\gamma}_n = (\sum_{i=1}^{n} x_i)/n$, then the probability that the estimation error is below ϵ is greater than our required confidence. Formally, we have: $Pr(|\bar{\gamma}_n - \gamma| \leq \epsilon) \geq 1 - \delta$. After the calculation of the number of samples n, a simple Monte Carlo simulation is performed [26].

Sequential Probability Ratio Test (SPRT). This sequential method [38] is a form of hypothesis testing, which can answer qualitative questions. Given a random variable X with a probability density function $f(x, \theta)$, we want to decide, whether a null hypothesis $H_0 : \theta = \theta_0$ or an alternative hypothesis $H_1 : \theta = \theta_1$ is true for desired type I and II errors $(\alpha,\ \beta)$. In order to make the decision, we start sampling and calculate the log-likelihood ratio after each observation of x_i:

$$\log \Lambda_m = \log \frac{p_1^m}{p_0^m} = \log \frac{\prod_{i=1}^{m} f(x_i, \theta_1)}{\prod_{i=1}^{m} f(x_i, \theta_0)} = \sum_{i=1}^{m} \log \frac{f(x_i, \theta_1)}{f(x_i, \theta_0)}$$

We continue sampling as long as $\log \frac{\beta}{1-\alpha} < \log \Lambda_m < \log \frac{1-\beta}{\alpha}$. H_1 is accepted when $\log \Lambda_m \geq \log \frac{1-\beta}{\alpha}$, and H_0 when $\log \Lambda_m \leq \log \frac{\beta}{1-\alpha}$ [18].

In this work, we form a hypothesis about the expected response time with the Monte Carlo method on the model. Then, we check with the SPRT if this hypothesis holds on a deployment of the SUT. This is faster than running a Monte Carlo simulation directly on the deployment, since the SPRT requires a far lower number of samples, i.e. test cases.

2.2 Property-Based Testing (PBT)

PBT is a random-testing technique that aims to check the correctness of properties. A property is a high-level specification of the expected behaviour of a function-under-test that should always hold. For example, the length of a concatenated list is always equal to the sum of lengths of its sub-lists:

$$\forall\, l_1, l_2 \in \mathit{Lists}[T] : \mathit{length}(\mathit{concatenate}(l_1, l_2)) = \mathit{length}(l_1) + \mathit{length}(l_2)$$

With PBT, we automatically generate inputs for such a property by applying data generators, e.g., the random list generator. The inputs are fed to the function-under-test and the property is evaluated. If it holds, then this indicates that the function works as expected, otherwise a counterexample is produced.

PBT also supports model-based testing. Models encoded as extended finite state machines (EFSMs) [23] can serve as source for state-machine properties. An EFSM is a 6-tuple (S, s_0, V, I, O, T). S is a finite set of states, $s_0 \in S$ is the initial state, V is a finite set of variables, I is a finite set of inputs, O is a finite set of outputs, T is a finite set of transitions. A transition $t \in T$ is a 5-tuple (s_s, i, g, op, s_t), s_s is the source state, i is an input, g is a guard, op is a sequence of output and assignment operations, s_t is the target state [23]. In order to derive a state-machine property from an EFSM, we have to write a specification comprising the initial state, commands and a generator for the next transition given the current state of the model. Commands encapsulate (1) preconditions that define the permitted transition sequences, (2) postconditions that specify the expected behaviour and (3) execution semantics of transitions for the model and the SUT. A state-machine property states that for all permitted transition sequences, the postcondition must hold after the execution of each transition resp. command [21,33]. Simply put, such properties can be defined as follows:

$$\begin{aligned}
&\mathit{cmd.runModel}, \mathit{cmd.runActual} : S \times I \rightarrow S \times O \\
&\mathit{cmd.pre} : I \times S \rightarrow \mathit{Boolean}, \mathit{cmd.post} : (S \times O) \times (S \times O) \rightarrow \mathit{Boolean} \\
&\forall s \in S, i \in I, \mathit{cmd} \in \mathit{Cmds} : \\
&\qquad \mathit{cmd.pre}(i, s) \implies \mathit{cmd.post}(\mathit{cmd.runModel}(i, s), \mathit{cmd.runActual}(i, s))
\end{aligned}$$

There are two functions to execute a command on the model and on the SUT: *cmd.runModel* and *cmd.runActual*. The precondition *cmd.pre* defines the valid inputs for a command. The postcondition *cmd.post* compares the outputs and states of the model and the SUT after the execution of a command.

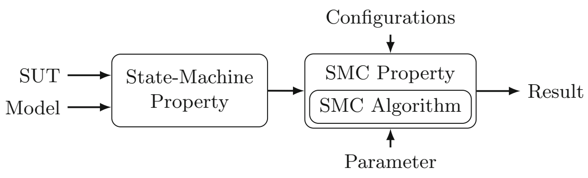


Fig. 2. Data flow diagram of an SMC property.

PBT is a powerful testing technique that allows a flexible definition of generators and properties via inheritance or composition. The first implementation of PBT was QuickCheck for Haskell [15]. Numerous reimplementations followed for other programming languages, like Hypothesis[1] for Python or ScalaCheck [30]. We developed our method with FsCheck[2]. FsCheck is a .NET port of QuickCheck with influences of ScalaCheck. It supports a functional programming style with F# and an object-oriented style with C#. We work with C#, since it is the programming language of our SUT.

2.3 Stochastic Timed Automata

Timed automata (TA) were originally introduced by Alur and Dill [6]. Several extensions of TA have been proposed, including stochastically enhanced TA [11] and continuous probabilistic TA [24]. We follow the definition of stochastic timed automata (STA) by Ballarini et al. [8]: An STA can be expressed as a tuple $(L, l_0, A, C, I, E, F, W)$, where the first part is a normal TA (L, l_0, A, C, I, E) and additionally it contains probability density functions $F = (f_l)_{l \in L}$ for the sojourn time and natural weights $W = (w_e)_{e \in E}$ for the edges. L is a finite set of locations, $l_0 \in L$ is the initial location, A is a finite set of actions, C is a finite set of clocks with real-valued valuations $u(c) \in \mathbb{R}_{>0}$, $I : L \mapsto \mathcal{B}(C)$ is a finite set of invariants for the locations and $E \subseteq L \times A \times \mathcal{B}(C) \times 2^C \times L$ is a finite set of transitions between locations, with an action, a guard and a set of clock resets. The transition relation can be described as follows. For a state (l, u), where $l \in L$ is a location and $u \in C \to \mathbb{R}_{\geq 0}$ is a clock valuation, the probability density functions f_l is used to choose the sojourn time d, which changes the state to $(l, u+d)$, where $u+d$ means that the clock valuation is changed $(u+d)(c) = u(c) + d$ for all $c \in C$. After this change, an edge e is selected out of the set of enabled edges $E(l, u+d)$ with the probability $w_e / \sum_{h \in E(l,u+d)} w_h$. Then, a transition to the target location l' of e and $u' = u + d$ is performed. For our models, the underlying stochastic process is a semi-Markov process since the clocks are reset at every transition, but we do not assume exponential delays, and therefore, it is not a standard continuous-time Markov chain.

[1] https://pypi.python.org/pypi/hypothesis
[2] https://fscheck.github.io/FsCheck

2.4 Integration of SMC into Property-Based Testing

We have demonstrated that SMC can be integrated into a PBT tool in order to perform SMC of PBT properties [3,5], which were explained in Sect. 2.2. These PBT properties can be evaluated on stochastic models, like in classical SMC, as well as on stochastic implementations. For the integration, we introduced our own new SMC properties, which take a PBT property, configurations for the PBT execution, and parameters for the specific SMC algorithm as input. Then, our properties perform an SMC algorithm by utilizing the PBT tool as simulation environment and they return either a quantitative or qualitative result, depending on the algorithm. Figure 2 shows how we can evaluate a state-machine property within an SMC property. Such a state-machine property can, e.g., be applied for a statistical conformance analysis by comparing an ideal model to a stochastic faulty implementation or it can also simulate a stochastic model. We evaluated our SMC properties by repeating case studies from the SMC literature and we were able to reproduce the results.

3 Method

In this section, we present the necessary steps to apply our method, based on the overview presented in Fig. 1. The steps are illustrated with an example of an incident manager that was introduced in our previous work [2,4].

Model-Based Testing. First, we perform model-based testing within PBT in order to produce log-data. This initial testing phase is conducted with a functional EFSM model. The SUT is a web-based tool that supports tasks, like creating, editing or closing incident objects, which can, e.g., be bug reports. These objects include attributes (form data) that are stored in a database. We automatically generate data for the attributes with PBT generators. An example model of the incident manager is illustrated in Fig. 3. This model is a hierarchical state machine [19]. There are sub-state machines for each incident object and select transitions can switch between these objects. We have a variable *activeObj* that identifies the currently open incident and a map (*stateMap*) that has an object identifier as key and stores the state of all incidents. Each sub-state machine shows the tasks that can be performed for an incident object. In reality, each of these tasks represents a page of the incident manager with required form fields (attributes). Hence, the transitions are parametrised with attributes [4].

With this functional model, we perform classical PBT, which generates random sequences of commands with form data. We run several testing processes concurrently in order to produce log-data that includes response times of simultaneous requests. Note that the tasks of the sub-state machines in Fig. 3 consist of multiple subtasks that are not shown in this figure for clarity. For example, there are subtasks for opening the page (*StartTask*), for setting attributes (*SetAttribute*), and for saving the page (*Commit*). Most of these subtasks are requests, hence, we record them in our logs. An example log from a non-productive test system with low computing resources (virtual machine) is represented in Table 1.

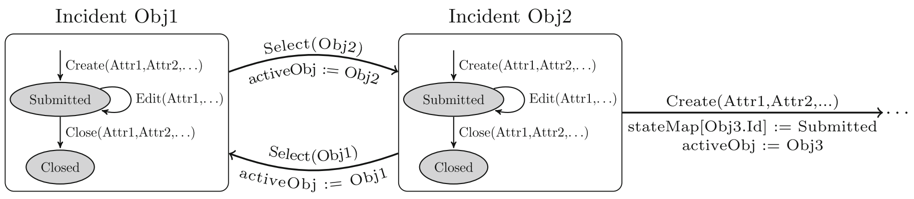


Fig. 3. Functional EFSM model of the incident manager [4].

Table 1. Example log-data of the incident manager.

Task	From	To	Subtask	#ActiveUsers	Attribute	ResponseTime[ms]
Create	Global	Submitted	StartTask	7	-	334
Create	Global	Submitted	SetAttribute	8	Assignee	77
Edit	Submitted	Submitted	StartTask	5	-	286
Create	Global	Submitted	Commit	6	-	918
Edit	Submitted	Submitted	SetAttribute	4	TestOrder	347

The log contains response times of tasks, subtasks, attributes, states (*From, To*) and simultaneous requests (*#ActiveUsers*). For this initial testing phase, we selected transitions with a uniform distribution.

Linear Regression. We apply a multiple linear regression in order to obtain distributions for response times. First, we performed data cleaning and preprocessing, where we, e.g., filter outliers. For example, we are not interested in unusual long response times that are caused by network disruptions. Our aim is to maximise the user satisfaction. Hence, we are mainly interested in average response times under normal conditions, but not in worst-case scenarios. Next, we select the log variables, which have a high influence on the response time. This phase is called *feature selection*. This and all other steps are facilitated with the help of data visualisations, e.g., scatter plots or correlation matrices. Finally, we can run the linear regression by applying R, which is a standard statistics tool.[3] A more detailed description of the regression can be found in our previous work [37] and in the work of Rencher and Christensen [34].

As a result, we obtain estimates of the mean response time and standard errors for the regression variables. We combine these values with a linear combination to obtain parameters for the normal distribution for specific variable assignments. This combination is done inside the function *rtime* (response time). This function takes a task, a subtask the number of active users and an optional attribute as input and returns the parameters μ and σ of the normal distribution.

[3] https://www.r-project.org

$$rtime : Task \times Subtask \times \mathbb{N}_{>0} \times Attribute \rightarrow \mathbb{R} \times \mathbb{R}$$

```
{TaskWeights: { Create: 70, Edit: 45, Close: 25, Select: 30 },
TaskWaitMin:500,TaskWaitMax:1500, SubTaskWaitMin:300,SubTaskWaitMax:500,
WaitPerReference: 10, WaitPerCharacter: 30 }
```

Listing 1.1: User profile in the JavaScript Object Notation (JSON) format.

Monte Carlo Simulation. In order to apply SMC for a realistic usage scenario, we integrate a given user profile and response-time distributions from the linear regression into the functional model. An example user profile for the incident manager is shown in Listing 1.1. It includes weights for tasks, user input (waiting) time intervals between tasks/subtasks that represent the time that a user needs for the input and data specific waiting factors, e.g., a delay per character, or a delay per reference for the number of options of a drop-down menu.

The extension of the initial functional model with a user profile and response-time distributions gives us a combined model that is a stochastic timed automaton (Sect. 2.3). Figure 4 illustrates this automaton for one incident object. Note, we only show the combined model of one sub-state machine of the hierarchical EFSM in Fig. 3 for brevity. All locations (states) l in this combined model include a sojourn time that is defined with a probability density function f_l. The tasks of the functional model where separated into subtasks in order to represent the response times of individual requests. Each subtask comprises an edge that calls the *rtime* function to receive the parameters (μ, σ) and a location (d_i) that applies these parameters in a normal distribution for f_l. All other locations describe f_l with a uniform distribution given by an upper and lower bound $[a, b]$. The locations *Submitted* and *Closed* have bounds from the user input time intervals between tasks of the user profile and for the other locations (w_i) the bounds are calculated in a separate edge with a function *utime* (user time). This function takes into account the user-time intervals between subtasks and the data-dependent time, e.g., the wait per character, from the user profile, and returns according bounds. The task weights of the user profile are attached to the edge weights w_e and they are shown before a edge name (in bold). It can be seen that each transition or task of the initial functional model is now represented as a sequence of edges with a silent edge at the end. Note, we also included the *select* tasks, which were explained earlier. A *create* task is also possible in the *submitted* and *closed* location, but we omit additional edges for this, in order to keep the figure simple. Note that we also omit parameters and their assignments for the *rtime* and *utime* functions. The parameters for *rtime* were already explained before and *utime* takes the generated attribute data as input and returns associated intervals for the user-input time.

With this combined model, we can evaluate user profiles by simulating their expected response times. Furthermore, we can analyse a user population consisting of multiple users, by running several models concurrently. We execute this model to answer questions, like "What is the probability that the response time of each *Commit* subtask of a user within a population is under a certain threshold?". Such questions can be answered with a Monte Carlo simulation with Chernoff-Hoeffding bound. For example, checking the probability for

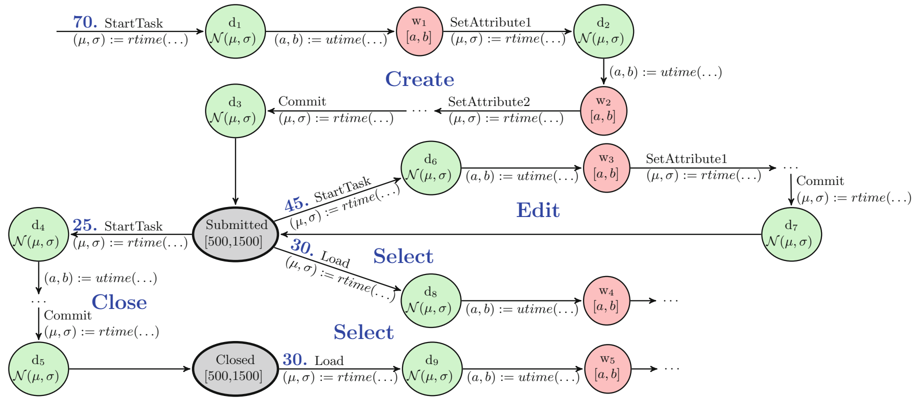


Fig. 4. Stochastic timed automata model for one incident object.

a response-time threshold of one second for each user of a population of 10 users with parameters $\epsilon = 0.05$ and $\delta = 0.01$, requires 1060 samples and returns a probability of 0.593, when a test-case length of three tasks is considered. Fortunately, the SPRT requires fewer samples and is, therefore, better suited for the evaluation of the reference SUT or SUT deployments.

Hypothesis Testing with the SPRT. The probability that was computed on the model with a Monte Carlo simulation serves as a hypothesis in order to check, if SUT deployments are at least as good as the reference SUT. However, first we evaluate the hypothesis on the reference SUT. We apply the computed probability as alternative hypothesis and select a probability of 0.493 as null hypothesis, which is 0.1 smaller, because we want to be able to reject the hypothesis that the SUT has a smaller probability. By running the SPRT (with 0.01 as type I and II error parameters) for each user of the population, we can check these hypotheses. The alternative hypothesis was accepted for all users and on average 76.8 samples were needed for the decision. After we have evaluated the hypotheses on the reference SUT, we can reuse the hypotheses to check if deployments of this SUT provide a similar performance. For example, an evaluation of a deployment with less RAM might result in the acceptance of the same hypotheses. The acceptance of the same hypotheses means that the deployment provides the same or a similar performance as the reference SUT for our usage scenario, otherwise the deployment has worse response times.

Note that our method was implemented in the same way, as described in our previous work [37], by introducing custom generators for the response- and user-input time. For brevity, we omit the details of the implementation.

4 Evaluation

System-Under-Test. We evaluated our method by applying it to a web-service application from the automotive domain, which was provided by our indus-

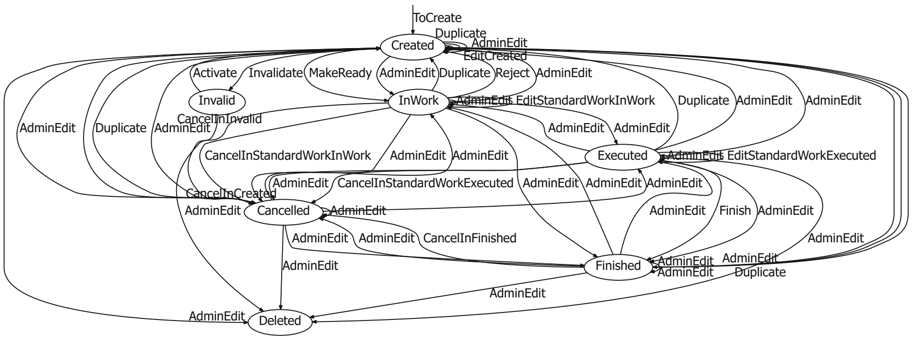


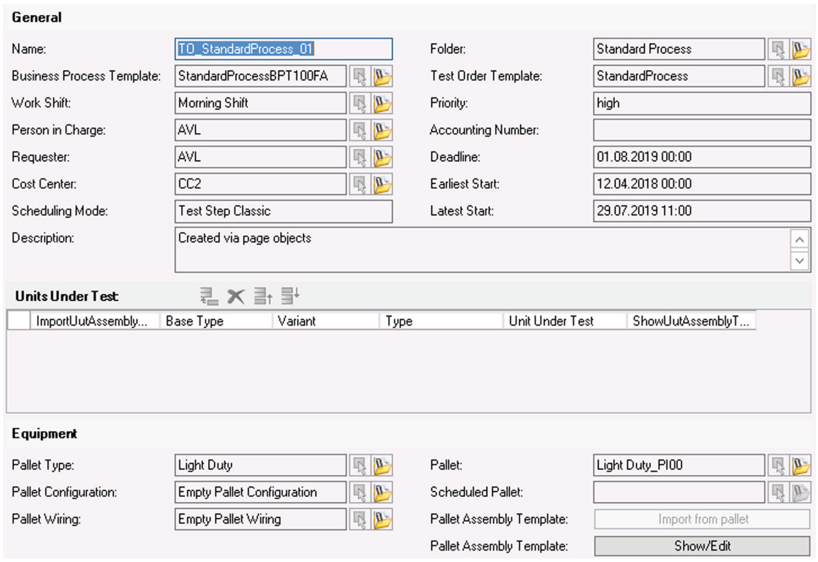
Fig. 5. Example sub-state machine of one test order object.

trial partner AVL[4]. The application is called Testfactory Management Suite (TFMS) and it enables various management activities of test beds, like test definition, planning, preparation, execution and data management/analysis for testing engines. TFMS is based on a client/server architecture. The server is connected to an external MongoDB database. Client and server communicate via (SOAP) web services hosted on a Microsoft Internet Information Server (IIS). There are several client types that support different activities, e.g., one client for managing test orders for test beds. As part of the software quality verification process, there is a test framework that simulates a client. This framework facilitates the creation of requests to the server, and hence, supports our testing method which works from a client's perspective [4].

TFMS consists of several modules which group together objects of the application domain and associated activities. For our evaluation, we focused on one main module, the Test Order Manager. This module enables the configuration and execution of test orders, which are basically a composition of steps that are necessary for a test sequence at an automotive test bed. Figure 5 shows an example sub-state machine for the tasks of one test order object. The complete model of the Test Order Manager is also a hierarchical state machine, like Fig. 3, but it is even more complex and therefore not presentable. Each task of the state machine represents the invocation of a page, entering data for form fields and saving the page, e.g., the page of one task is shown in Fig. 6. The Test Order Manager contains further sub-state machines for the creation of test orders, like Business Process Templates, but they are similar to this state machine, and are therefore omitted [2].

Test Setup. We evaluated a TFMS server (version 1.8) that was running on a virtual machine with Windows Server 2012. Note that the example given in Sect. 3, was done with TFMS 1.7. Our reference SUT (D_0) had 15 GB of RAM and 7 Intel Xeon E5-2690v4 2.6 GHz CPUs. A similar virtual machine with 6 GB RAM and 3 CPUs was used to run the test clients. We defined a set

[4] https://www.avl.com



General

Name:	TO_StandardProcess_01	Folder:	Standard Process
Business Process Template:	StandardProcessBPT100FA	Test Order Template:	StandardProcess
Work Shift:	Morning Shift	Priority:	high
Person in Charge:	AVL	Accounting Number:	
Requester:	AVL	Deadline:	01.08.2019 00:00
Cost Center:	CC2	Earliest Start:	12.04.2018 00:00
Scheduling Mode:	Test Step Classic	Latest Start:	29.07.2019 11:00
Description:	Created via page objects		

Units Under Test

	ImportUutAssembly...	Base Type	Variant	Type	Unit Under Test	ShowUutAssemblyT...

Equipment

Pallet Type:	Light Duty	Pallet:	Light Duty_PI00
Pallet Configuration:	Empty Pallet Configuration	Scheduled Pallet:	
Pallet Wiring:	Empty Pallet Wiring	Pallet Assembly Template:	Import from pallet
		Pallet Assembly Template:	Show/Edit

Fig. 6. Screen shot of a Test Order Manager web form for one task.

of deployments by varying values for the CPUs, the RAM size, the network bandwidth, and the network delay. These deployments (D_i) are shown in Table 2. Since the server was running on a virtual machine, the hardware settings could easily be changed. A tool called Network Emulator for Windows helped us to configure the network setup of the test client, e.g., it allowed us to decrease the network bandwidth. The testing phase and also the model evaluation were performed with the PBT tool FsCheck 2.8.2.

Monte Carlo Simulation. We applied our method in order to answer the following question: "What is the probability that the response time of all requests within a task sequence of a fixed length, i.e. a test case, is under a specific threshold for each user within a population?". For this evaluation, a user profile was created in cooperation with domain experts from AVL. This profile was similar to the one of Listing 1.1, and is hence, omitted. Also the stochastic timed automata model was similar to that of Fig. 4, but more complex, since the Test Order Manager comprises multiple sub-state machines for different object types. We also omit this model for brevity. We applied the model in the same way as described in Sect. 3, in order to evaluate user populations of different sizes, and we checked various response-time thresholds with a fixed test-case size of four tasks.

The model was analysed with a Monte Carlo simulation with Chernoff-Hoeffding bound with parameters $\epsilon = 0.05$ and $\delta = 0.01$, which requires 1060 samples (per data point). Figure 7 shows the results. As expected, a decrease in the probability of our given question can be observed, when the number of users increases or the threshold decreases. Note that an advantage of the evaluation

Table 2. Different system deployments with various hardware/network settings.

Deplyoment	Hardware		Network	
	#CPUs	RAM [GB]	Bandwidth [Mbps]	Delay [ms]
D_0	7	15	1000	0
D_1	7	4	1000	0
D_2	2	15	1000	0
D_3	7	15	500	0
D_4	7	15	100	0
D_5	7	15	50	0
D_6	7	15	1000	25
D_7	7	15	1000	10

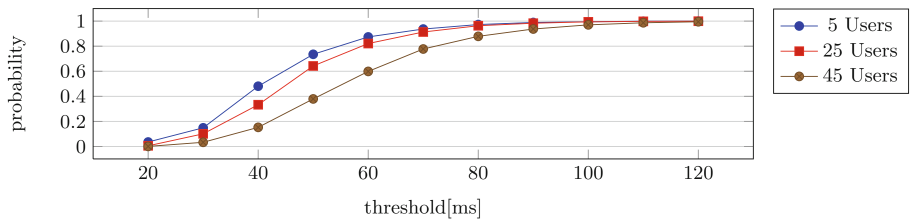


Fig. 7. Test Order Manager Monte Carlo simulation results of the model.

of the model is that the model execution can be accelerated with a virtual time. We apply a virtual time of 1/10 of the actual time, which speeds up the model execution by a factor of ten (compared to the SUT).

Hypothesis Testing with the SPRT. Next, we applied the probabilities of the Monte Carlo simulation as hypotheses (H_1) for SPRTs of the different deployments. We selected six data points of Fig. 7 with interesting thresholds and different user numbers in order to form the hypotheses shown in Table 3. We evaluate all deployments as explained in Sect. 3 by applying the SPRT with the same parameters. Figure 8 summarises the results in three groups: one for the deployments (and SPRTs), where all clients accepted H_1, one where there was no clear consensus among the clients, and one where all clients accepted H_0. It can be seen that H_1 was accepted by most of the deployments, which means that they provide a similar performance. For one deployment (D_5) only SPRT 1–4 were successful, SPRT 5–6 were inconclusive, i.e. 48 % of the clients accepted H_1 for SPRT 5 and 44 % for SPRT 6. For two deployments, H_0 was accepted, which means that their response time was worse than that of the reference SUT. In summary, it can be said that a change in the server hardware did not significantly affect the performance, as H_1 was accepted for all deployments with a changed hardware. Also, a change in the network bandwidth had only a weak influence

Table 3. Different SPRTs for various numbers of users and thresholds.

SPRT No.	#Users	Threshold [ms]	H_0	H_1
1	5	50	0.478	0.729
2	25	50	0.400	0.650
3	45	50	0.201	0.451
4	5	100	0.746	0.996
5	25	100	0.744	0.994
6	45	100	0.738	0.988

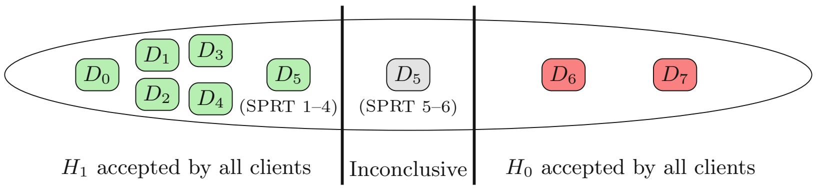


Fig. 8. SPRT results of the different deployments.

on the performance. A clear change in the performance was only observed for deployments with a higher network delay.

Additionally, we evaluated the number of needed samples of the SPRTs. Note that in order to obtain an average number of needed samples, we run the SPRT concurrently for each user of the population and calculate the average of these runs. Multiple independent SPRT runs would produce a better average, but the computation time was too high. Figure 9 shows the average number of needed samples for the SPRTs of different deployments. It can be seen that certain SPRTs are quite easy to check, e.g., SPRT 3 only needs about 6–13 samples, other SPRTs take more than twice as many samples. However, a maximum of about 30 samples is still very low compared to the 1060 samples of the Monte

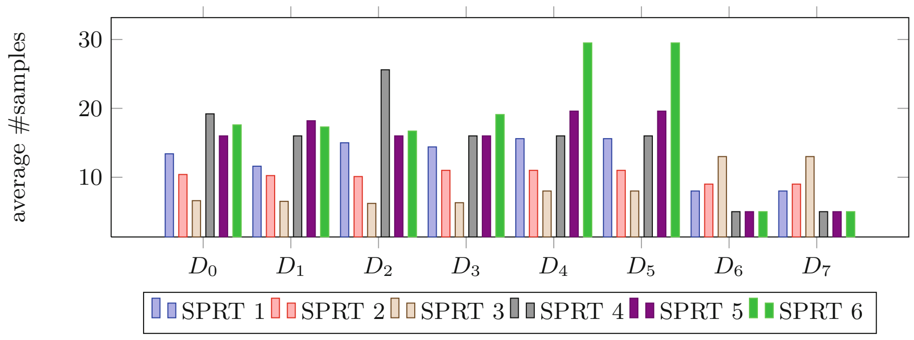


Fig. 9. Average number of samples (test cases) for the SPRTs of our deployments.

Carlo simulation. This low number of samples allows us to evaluate multiple SUT deployments within a feasible time.

5 Conclusion

We have demonstrated that we can apply SMC with learned timed models in order to answer questions about the expected response time of given usage scenarios, like "What is the probability that the response time of each user within a population, is under a specific threshold?". Moreover, we have illustrated that we can verify the results of such evaluations with hypothesis testing on a real system. Additionally, we checked deployments of an SUT by reusing the hypotheses of a reference SUT.

A major benefit of our approach is that it enables an efficient performance comparison of a reference system with system deployments for specific usage scenarios. This is especially helpful, when customer recommendation for the hardware or network settings are needed for a deployment that should satisfy certain user expectations. Another advantage of our method is that it is realised within a PBT tool, which increases the accessibility for testers from industry, because the models and properties can be defined in a high-level programming language. Hence, there is no need to learn new notations.

We have evaluated our method with an industrial case study of a web-service application, and it showed promising results. We analysed various deployments of an SUT with different hardware and network settings. This analysis showed that deployments with different server hardware provide a comparable performance as the reference system for our given usage scenarios. Only deployments with higher network delays showed a significant performance loss.

In the future, we plan to combine our technique with different learning methods. Since a linear regression requires still a high manual effort, we aim to evaluate learning methods that support a higher degree of automation. Moreover, an analysis of the applicability of our method for other performance indicators than response times, e.g., for energy, has a great potential for future work.

Acknowledgements. This research was funded by the Austrian Research Promotion Agency (FFG), project TRUCONF, No. 845582. We are grateful to the project team and to the anonymous reviewers for their remarks.

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1), 6:1–6:39 (2018)
2. Aichernig, B.K., Schumi, R.: Property-based testing with FsCheck by deriving properties from business rule models. In: ICSTW, pp. 219–228. IEEE (2016)
3. Aichernig, B.K., Schumi, R.: Towards integrating statistical model checking into property-based testing. In: MEMOCODE, pp. 71–76. IEEE (2016)

4. Aichernig, B.K., Schumi, R.: Property-based testing of web services by deriving properties from business-rule models. Softw. Syst. Model. 1–23 (2017). https://doi.org/10.1007/s10270-017-0647-0
5. Aichernig, B.K., Schumi, R.: Statistical model checking meets property-based testing. In: ICST, pp. 390–400. IEEE (2017)
6. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994)
7. Arts, T.: On shrinking randomly generated load tests. In: Erlang 2014, pp. 25–31. ACM (2014)
8. Ballarini, P., Bertrand, N., Horváth, A., Paolieri, M., Vicario, E.: Transient analysis of networks of stochastic timed automata using stochastic state classes. In: Joshi, K., Siegle, M., Stoelinga, M., D'Argenio, P.R. (eds.) QEST 2013. LNCS, vol. 8054, pp. 355–371. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40196-1_30
9. Banga, G., Druschel, P.: Measuring the capacity of a web server under realistic loads. World Wide Web **2**(1–2), 69–83 (1999)
10. Becker, S., Koziolek, H., Reussner, R.H.: The Palladio component model for model-driven performance prediction. J. Syst. Softw. **82**(1), 3–22 (2009)
11. Blair, L., Jones, T., Blair, G.: Stochastically enhanced timed automata. In: Smith, S.F., Talcott, C.L. (eds.) FMOODS 2000. IAICT, vol. 49, pp. 327–347. Springer, Boston, MA (2000). https://doi.org/10.1007/978-0-387-35520-7_17
12. Book, M., Gruhn, V., Hülder, M., Köhler, A., Kriegel, A.: Cost and response time simulation for web-based applications on mobile channels. In: QSIC, pp. 83–90. IEEE (2005)
13. Bulychev, P.E., et al.: UPPAAL-SMC: statistical model checking for priced timed automata. In: QAPL. EPTCS, vol. 85, pp. 1–16. Open Publishing Association (2012)
14. Chen, X., Mohapatra, P., Chen, H.: An admission control scheme for predictable server response time for web accesses. In: WWW, pp. 545–554. ACM (2001)
15. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: ICFP, pp. 268–279. ACM (2000)
16. Claessen, K., et al.: Finding race conditions in Erlang with QuickCheck and PULSE. In: ICFP, pp. 149–160. ACM (2009)
17. Draheim, D., Grundy, J.C., Hosking, J.G., Lutteroth, C., Weber, G.: Realistic load testing of web applications. In: CSMR, pp. 57–70. IEEE (2006)
18. Govindarajulu, Z.: Sequential Statistics. World Scientific (2004)
19. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
20. Hoeffding, W.: Probability inequalities for sums of bounded random variables. J. Am. Stat. Assoc. **58**(301), 13–30 (1963)
21. Hughes, J.: QuickCheck testing for fun and profit. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 1–32. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-69611-7_1
22. Hughes, J., Pierce, B.C., Arts, T., Norell, U.: Mysteries of Dropbox: property-based testing of a distributed synchronization service. In: ICST, pp. 135–145. IEEE (2016)
23. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine. In: ICST, pp. 230–239. IEEE (2009)

24. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Verifying quantitative properties of continuous probabilistic timed automata. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 123–137. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_11
25. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: an overview. In: Barringer, H., et al. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16612-9_11
26. Legay, A., Sedwards, S.: On statistical model checking with PLASMA. In: TASE, pp. 139–145. IEEE (2014)
27. Lu, Y., Nolte, T., Bate, I., Cucu-Grosjean, L.: A statistical response-time analysis of real-time embedded systems. In: RTSS, pp. 351–362. IEEE (2012)
28. Malik, H., Shakshuki, E.M.: Classification of post-deployment performance diagnostic techniques for large-scale software systems. Procedia Comput. Sci. **37**, 244–251 (2014)
29. Menascé, D.A.: Load testing of web sites. IEEE Internet Comput. **6**(4), 70–74 (2002)
30. Nilsson, R.: ScalaCheck: The Definitive Guide. IT Pro, Artima Incorporated (2014)
31. Norell, U., Svensson, H., Arts, T.: Testing blocking operations with QuickCheck's component library. In: Erlang 2013, pp. 87–92. ACM (2013)
32. Nourikhah, H., Akbari, M.K., Kalantari, M.: Modeling and predicting measured response time of cloud-based web services using long-memory time series. J. Supercomput. **71**(2), 673–696 (2015)
33. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Erlang 2011, pp. 39–50. ACM (2011)
34. Rencher, A., Christensen, W.: Methods of Multivariate Analysis. Wiley, Hoboken (2012)
35. Rina, S.T.: A comparative study of performance testing tools. Int. J. Adv. Res. Comp. Sci. Softw. Eng. IJARCSSE **3**(5), 1300–1307 (2013)
36. Roloff, E., Diener, M., Carissimi, A., Navaux, P.O.A.: High performance computing in the cloud: Deployment, performance and cost efficiency. In: CloudCom, pp. 371–378. IEEE Computer Society (2012)
37. Schumi, R., Lang, P., Aichernig, B.K., Krenn, W., Schlick, R.: Checking response-time properties of web-service applications under stochastic user profiles. In: Yevtushenko, N., Cavalli, A.R., Yenigün, H. (eds.) ICTSS 2017. LNCS, vol. 10533, pp. 293–310. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67549-7_18
38. Wald, A.: Sequential Analysis. Courier Corporation (1973)
39. Yu, J., Han, J., Schneider, J., Hine, C.M., Versteeg, S.: A Petri-Net-based virtual deployment testing environment for enterprise software systems. Comput. J. **60**(1), 27–44 (2017)
40. Zhang, F., et al.: Modeling and evaluation of wireless sensor network protocols by stochastic timed automata. Electr. Notes Theor. Comput. Sci. **296**, 261–277 (2013)

Probabilistic Analysis of Timing Constraints in Autonomous Automotive Systems Using Simulink Design Verifier

Eun-Young Kang[1,2](✉) and Li Huang[2]

[1] University of Namur, Namur, Belgium
eykang@unamur.be
[2] School of Data and Computer Science, Sun Yat-Sen University, Guangzhou, China
huangl223@mail2.sysu.edu.cn

Abstract. Modeling and analysis of timing constraints is crucial in automotive systems. East-adl is a domain specific architectural language dedicated to safety-critical automotive embedded system design. In most cases, a bounded number of violations of timing constraints in systems would not lead to system failures when the results of the violations are negligible, called Weakly-Hard (WH). We have previously specified East-adl timing constraints in Clock Constraint Specification Language (Ccsl) and transformed timed behaviors in Ccsl into formal models amenable to model checking. Previous work is extended in this paper by including support for probabilistic analysis of timing constraints in the context of WH: Probabilistic extension of Ccsl, called PrCcsl, is defined and the East-adl timing constraints with stochastic properties are specified in PrCcsl. The semantics of the extended constraints in PrCcsl is translated into *Proof Objective Models* that can be verified using Simulink Design Verifier. Furthermore, a set of mapping rules is proposed to facilitate guarantee of translation. Our approach is demonstrated on an autonomous traffic sign recognition vehicle case study.

Keywords: East-adl · Timing constraints · Probabilistic Ccsl
Simulink Design Verifier · Weakly-hard system

1 Introduction

Software development for Cyber-Physical Systems (CPS) requires both functional and non-functional quality assurance to guarantee that CPS operate in a safety-critical context under timing constraints. Automotive electric/electronic systems are ideal examples of CPS in which the software controllers interact with physical environments. The continuous time behaviors of those systems often rely on complex dynamics as well as on stochastic behaviors. Formal verification and validation (V&V) technologies are indispensable and highly recommended for development of safe and reliable automotive systems [3,5].

X. Feng et al. (Eds.): SETTA 2018, LNCS 10998, pp. 170–186, 2018.
https://doi.org/10.1007/978-3-319-99933-3_12

Conventional formal analysis of timing models addresses worst case designs, typically used for hard deadlines in safety critical systems, however, there is great incentive to include "less-than-worst-case" designs with a view to improving efficiency without affecting the quality of timing analysis in the systems. The challenge is the definition of suitable model semantics providing reliable predictions of *system timing*, given the timing of individual components and their compositions. While the standard worst case models are well understood in this respect, the behavior and the expressiveness of "less-than-worst-case" models is far less investigated. In most cases, a bounded number of violations of timing constraints in systems would not lead to system failures when the results of the violations are negligible, called Weakly-Hard (WH) [9, 27]. In this paper, we propose a formal probabilistic modeling and analysis technique by extending the known concept of WH constraints to what is called "typical" worst case model and analysis.

East-adl (Electronics Architecture and Software Technology - Architecture Description Language) [4,6], aligned with AUTOSAR (Automotive Open System Architecture) standard [1], is the model-based development approach for the architectural modeling of safety-critical automotive embedded systems. A system in East-adl is described by `Functional Architectures (FA)` at different abstraction levels. The `FA` are composed of a number of interconnected *functionprototypes* (f_p), and the f_ps have ports and connectors for communication. East-adl relies on external tools for the analysis of specifications related to requirements. For example, behavioral description in East-adl is captured in external tools, i.e., Simulink/Stateflow [31]. The latest release of East-adl has adopted the time model proposed in the Timing Augmented Description Language (Tadl2) [10]. Tadl2 expresses and composes the basic timing constraints, i.e., repetition rates, end-to-end delays, and synchronization constraints. The time model of Tadl2 specializes the time model of MARTE, the UML profile for Modeling and Analysis of Real-Time and Embedded systems [28]. MARTE provides Ccsl, a time model and a Clock Constraint Specification Language, that supports specification of both logical and dense timing constraints for MARTE models, as well as functional causality constraints [24].

We have previously specified non-functional properties (timing and energy constraints) of automotive systems specified in East-adl and MARTE/Ccsl, and proved the correctness of specification by mapping the semantics of the constraints into Uppaal models for model checking [22]. Previous work is extended in this paper by including support for probabilistic analysis of timing constraints of automotive systems in the context WH: 1. Probabilistic extension of Ccsl, called PrCcsl, is defined and the East-adl/Tadl2 timing constraints with stochastic properties are specified in PrCcsl; 2. The semantics of the extended constraints in PrCcsl is translated into verifiable *Proof Objective Models* (POMs) for formal verification using Simulink Design Verifier (SDV) [2]; 3. A set of mapping rules is proposed to facilitate guarantee of translation. Our approach is demonstrated on an autonomous traffic sign recognition vehicle (AV) case study.

The paper is organized as follows: Sect. 2 presents an overview of Ccsl, Simulink/Stateflow and SDV. The AV is introduced as a running example in Sect. 3. Section 4 presents the formal definition of PrCcsl. Section 5 describes a set of translation patterns from Ccsl/PrCcsl to POMs and how our approaches provide support for formal analysis at the design level. The applicability of our method is demonstrated by performing verification on the AV case study in Sect. 6. Sections 7 and 8 present related work and the conclusion.

2 Preliminary

In our framework, we consider a subset of Ccsl and its extension with stochastic properties that is sufficient to specify East-adl timing constraints in the context of WH automotive systems. Simulink and Embedded Matlab (EML) are utilized for modeling purposes, and V&V are performed by the Simulink built-in verification tool, Simulink Design Verifier (SDV).

Clock Constraint Specification Language (Ccsl): Ccsl [8,24] clocks describe events in a system and measure occurrences of the events. The physical time is represented by a dense clock (with a base) and discretized into a logical clock. *idealClock* is a predefined dense clock whose unit is *second*. We define a universal clock *ms* based on *idealClock*: ms = *idealClock* `discretizedBy` 0.001, where *ms* is a periodic clock that ticks every 1 ms. A step is a tick of the universal clock. Hence the length of one step is 1 ms in this paper.

Ccsl provides two types of clock constraints, *relation* and *expression*: A *relation* limits the occurrences among different events/clocks. Let C be a set of clocks, $c1, c2 \in C$, `Coincidence` relation ($c1 \equiv c2$) specifies that two clocks must tick simultaneously. `Precedence` relation ($c1 \prec c2$) limits that $c1$ runs faster than $c2$, i.e., $\forall k \in \mathbb{N}^+$, where $\mathbb{N}^+$ is the set of positive natural numbers, the k^{th} tick of $c1$ must occur prior to the k^{th} tick of $c2$. `Causality` relation ($c1 \preceq c2$) represents a relaxed version of `Precedence`, allowing the two clocks to tick at the same time. `Subclock` ($c1 \subseteq c2$) indicates the relation between two clocks, *superclock* ($c1$) and *subclock* ($c2$), s.t. each tick of the subclock must correspond to a tick of its superclock at the same step. `Exclusion` ($c1 \# c2$) prevents the instants of two clocks from being coincident. An *expression* derives new clocks from the already defined clocks: `PeriodicOn` builds a new clock found on a *base* clock and a *period* parameter, s.t., the instants of the new clocks are separated by a number of instants of the *base* clock. The number is given as *period*. `DelayFor` results in a clock by delaying the *base* clock for a given number of ticks of a *reference* clock. `Infimum`, denoted `Inf`, is defined as the slowest clock that is faster than both $c1$ and $c2$. `Supremum`, denoted `Sup`, is defined as the fastest clock that is slower than $c1$ and $c2$.

Simulink and SDV: Simulink [31] is a synchronous data flow language, which provides different types of *blocks* for modeling and simulation of dynamic systems and code generation. Simulink supports the definition of custom blocks via Stateflow diagrams or *user-defined function* blocks written in EML, C, and

C++. SDV is a formal verification tool that performs reachability analysis on Simulink/Stateflow (S/S) model with Prover plugin. The satisfiability of each reachable state is determined by a SAT solver. A proof objective model is specified in Simulink/SDV and illustrated in Fig. 1. A set of data (predicates) on the input flows of *System* is constrained via ≪Proof Assumption≫ blocks during proof construction. A set of proof objectives are constructed via a function F block and the output of F is specified as input to a property P block. P passes its output signal to an ≪Assertion≫ block and returns *true* when the predicates set on the input data flows of the outline model are satisfied. Whenever ≪Assertion≫ is utilized, SDV verifies whether the specified input data flow is always *true*. Any failed proof attempt ends in the generation of a counterexample representing an execution path to an invalid state. A harness model is generated to analyze the counterexample and refine the model.

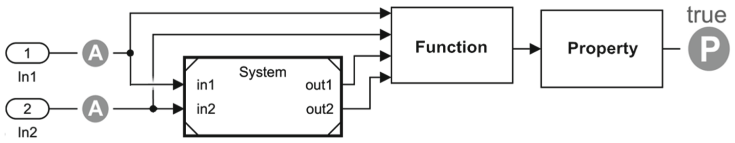


Fig. 1. General verification models in SDV

3 Running Example: Traffic Sign Recognition Vehicle

An autonomous vehicle (AV) [20,21] application using Traffic Sign Recognition is adopted to illustrate our approach. The AV reads the road signs, e.g., "speed limit" or "right/left turn", and adjusts speed and movement accordingly. The functionality of AV, augmented with timing constraints and viewed as `Functional Design Architecture (FDA)` (`designFunctionTypes`), consists of the following f_ps in Fig. 2: `System` function type contains four f_ps, i.e., the `Camera` captures sign images and relays the images to `SignRecognition` periodically. `Sign Recognition` analyzes each frame of the detected images and computes the desired images (sign types). `Controller` determines how the speed of the vehicle is adjusted based on the sign types and the current speed of the vehicle. `VehicleDynamic` specifies the kinematics behaviors of the vehicle. `Environment` function type consists of three f_ps, i.e., the information of traffic signs, random obstacles, and speed changes caused by environmental influences described in `TrafficSign`, `Obstacle`, and `Speed` f_ps respectively.

We consider the `Periodic, Execution, End-to-End, Synchronization, Sporadic`, and `Comparison` timing constraints on top of the AV East-ADL model, which are sufficient to capture the constraints described in Fig. 2. Furthermore, we extend East-ADL/Tadl2 with an `Exclusion` timing constraint (R8 in Fig. 2) that integrates relevant concepts from the Ccsl constraint, i.e., two events cannot occur simultaneously.

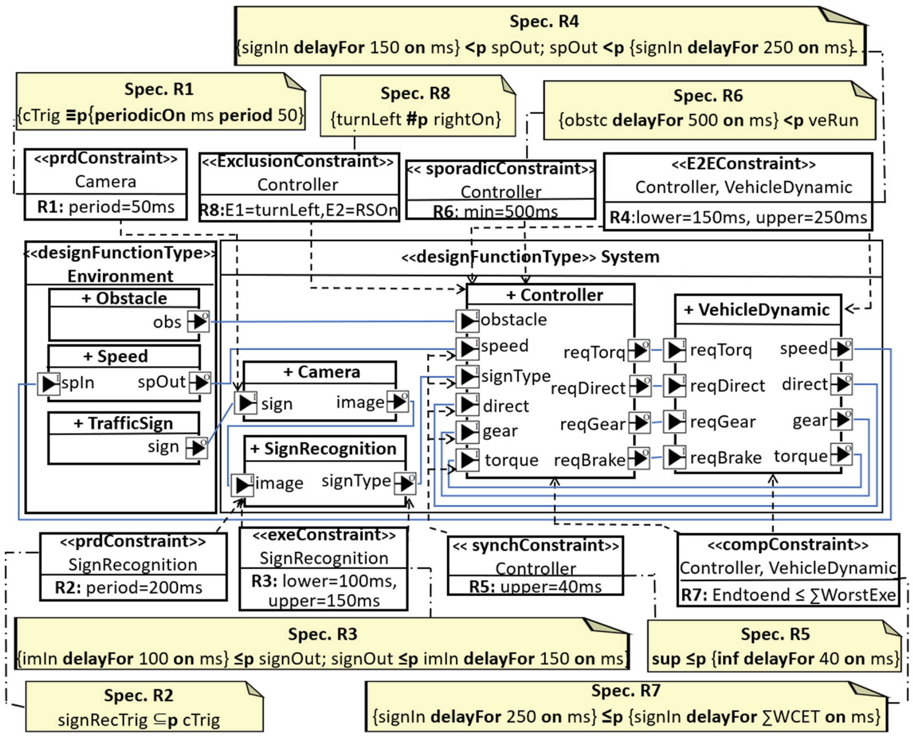

Spec. R4
{signIn delayFor 150 on ms} <p spOut; spOut <p {signIn delayFor 250 on ms}
Spec. R1
{cTrig ≡p{periodicOn ms period 50}
Spec. R8
{turnLeft #p rightOn}
Spec. R6
{obstc delayFor 500 on ms} <p veRun
<<prdConstraint>> Camera
R1: period=50ms
<<ExclusionConstraint>> Controller
R8:E1=turnLeft,E2=RSOn
<< sporadicConstraint>> Controller
R6: min=500ms
<<E2EConstraint>> Controller, VehicleDynamic
R4:lower=150ms, upper=250ms
<<designFunctionType>> Environment
<<designFunctionType>> System
+ Obstacle
obs
+ Speed
spIn
spOut
+ TrafficSign
sign
+ Camera
sign
image
+ SignRecognition
image
signType
+ Controller
obstacle
speed
signType
direct
gear
torque
reqTorq
reqDirect
reqGear
reqBrake
+ VehicleDynamic
reqTorq
reqDirect
reqGear
reqBrake
speed
direct
gear
torque
<<prdConstraint>> SignRecognition
R2: period=200ms
<<exeConstraint>> SignRecognition
R3: lower=100ms, upper=150ms
<< synchConstraint>> Controller
R5: upper=40ms
<<compConstraint>> Controller, VehicleDynamic
R7: Endtoend ≤ ∑WorstExe
Spec. R3
{imIn delayFor 100 on ms} ≤p signOut; signOut ≤p imIn delayFor 150 on ms}
Spec. R5
sup ≤p {inf delayFor 40 on ms}
Spec. R2
signRecTrig ⊆p cTrig
Spec. R7
{signIn delayFor 250 on ms} ≤p {signIn delayFor ∑WCET on ms}

Fig. 2. AV in EAST-ADL augmented with TADL2 timing constraints (`R.ID`), specified in PrCCSL (`Spec.R.ID`)

R1. The camera must capture an image every 50 ms. In other words, a `Periodic` acquisition of `Camera` must be carried out every 50 ms.

R2. The captured image must be recognized by an AV every 200 ms, i.e., a `Periodic` constraint on `SignRecognition` f_p.

R3. The detected image should be computed within [100, 150] ms in order to generate the desired sign type, the `SignRecognition` must complete its execution within [100, 150] ms.

R4. When a traffic sign is recognized, the speed of AV should be updated within [150, 250] ms. An `End-to-End` constraint on `Controller` and `VehicleDynamic`, i.e., the time interval from the input of `Controller` to the output of `VehicleDynamic` must be within a certain time.

R5. The required environmental information should arrive to the controller within 40 ms. Input signals (`speed`, `signType`, `direct`, `gear` and `torque` ports) must be detected by `Controller` within a given time window, i.e., the tolerated maximum constraint is 40 ms.

R6. If the mode of AV switches to “emergency stop” due to a certain obstacle, it should not revert back to “automatic running” mode within a specific time period. That is interpreted as a `Sporadic` constraint, i.e., the mode of AV is changed to `Stop` because of the encounter with an obstacle and it should not revert back to `Run` mode within 500 ms.

R7. The execution time interval from `Controller` to `VehicleDynamic` must be less than or equal to the sum of the worst case execution time interval of each f_p.

R8. While AV turns left, the “turning right” mode should not be activated. The events of turning left and right are considered as exclusive and specified as an `Exclusion` constraint.

`Delay` constraint gives duration bounds (minimum and maximum) between two events *source* and *target*. This is specified using *lower, upper* values given as either `Execution` constraint (R3) or `End-to-End` constraint (R4). `Synchronization` constraint describes how tightly the occurrences of a group of events follow each other. All events must occur within a sliding window, specified by the *tolerance* attribute, i.e., the maximum time interval allowed between events (R5). `Periodic` constraint states that the period of successive occurrences of a single event must have a time interval (R1–R2). `Sporadic` constraint states that *events* can arrive at arbitrary points in time, but with defined minimum inter-arrival times between two consecutive occurrences (R6). `Comparison` constraint delimits that two consecutive occurrences of an event should have a minimum inter-arrival time (R7). `Exclusion` constraint states that two events must not occur at the same time (R8). Those timing constraints are formally specified (seen as `Spec. R. IDs` in Fig. 2) using clock *relation* and *expression* in the context of WH then verified utilizing probabilistic analysis techniques that are described further in the following sections.

4 Probabilistic Extension of *Relations* in CCSL

To perform the formal specification and probabilistic verification of East-adl timing constraints (R1 – R8 in Sect. 3.), Ccsl *relations* are augmented with probabilistic properties, called PrCcsl, based on WH [9]. To describe the bound on the number of allowed constraint violations in WH, we extend Ccsl *relations* with a probabilistic parameter p, where p is the probability threshold. PrCcsl is satisfied if and only if the probability of *relation* constraint being satisfied is greater than or equal to p.

Definition 1 (PrCCSL). *Let $c1$, $c2$ and $\mathcal{M}$ be two logical clocks and a system model. The probabilistic extension of relation constraints, denoted $c1 \sim_p c2$, is satisfied if the following condition holds:*

$$\mathcal{M} \vDash c1 \sim_p c2 \iff Pr(c1 \sim c2) \geq p$$

where $\sim \in \{\subseteq, \equiv, \prec, \preceq, \#\}$, $Pr(c1 \sim c2)$ is the probability of the relation $c1 \sim c2$ being satisfied, and $p \in [0, 1]$ is the probability threshold.

$Pr(c1 \sim c2)$ is calculated based on clock ticks: $Pr(c1 \sim c2) = \frac{m}{k}$, where k is the total number of ticks and m is a number of ticks satisfying the clock relation $c1 \sim c2$.

Definition 2 ***(Tick and History).*** *For $c \in C$, the tick of c is indicated by a function t_c: $\mathbb{N} \rightarrow \{0, 1\}$. For $i \in \mathbb{N}$, $t_c(i)$ is a boolean variable that indicates whether c ticks at the i^{th} step, which is defined as: if c ticks at step i, $t_c(i) = 1$; otherwise $t_c(i) = 0$. The history of c is a function h_c: $\mathbb{N} \rightarrow \mathbb{N}$. $h_c(i)$ that represents the number of ticks of c that have been fired prior to the i^{th} step, which can be defined as: (1) $h_c(0) = 0$; (2) $\forall\ i \in \mathbb{N}^+,\ t_c(i) = 0 \implies h_c(i+1) = h_c(i)$; (3) $\forall\ i \in \mathbb{N}^+,\ t_c(i) = 1 \implies h_c(i+1) = h_c(i) + 1$.*

The five CCSL *relations*, `Subclock`, `Coincidence`, `Exclusion`, `Causality` and `Precedence`, are considered and the related probabilistic extensions are defined.

Definition 3 (Probabilistic Subclock). *The probability of subclock relation between $c1$ and $c2$, denoted $c1 \subseteq_p c2$, is satisfied if the following conditions hold:*

$$\mathcal{M} \vDash c1 \subseteq_p c2 \iff Pr(c1 \subseteq c2) \geq p$$

where $Pr(c1 \subseteq c2) = \frac{m}{k}$, $k = \sum_{i=0}^{n} t_{c1}(i)$, $m = \sum_{i=0}^{n} \{t_{c1}(i) \wedge (t_{c1}(i) \implies t_{c2}(i))\}$

n refers to the simulation bound (number of steps of an execution). k is the total number of ticks of the subclock $c1$ during the execution. m is the number of ticks of $c1$ satisfying the `subclock` relation. A tick of the subclock $c1$ satisfies the relation if at the step it occurs, its superclock $c2$ ticks. An example is shown in Fig. 3: among the 30 steps, $c1$ ticks seven times, and six of them (denoted by the arrows) satisfy `subclock` relation. In this case, $n = 30$, $k = 7$ and $m = 6$.

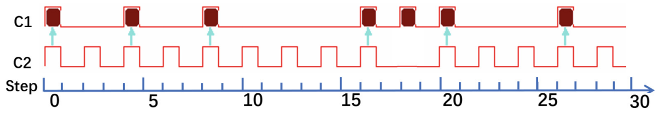


Fig. 3. Example of subclock relation

`Coincidence` relation states that two clocks should tick at the same step. i.e., they are subclocks of each other.

Definition 4 (Probabilistic Coincidence). *The probability of coincidence relation between $c1$ and $c2$, denoted $c1 \equiv_p c2$, is satisfied if the following conditions hold:*

$$\mathcal{M} \vDash c1 \equiv_p c2 \iff Pr(c1 \equiv c2) \geq p$$

where $Pr(c1 \equiv c2) = \frac{m}{k}$, $k = \sum_{i=0}^{n} \{t_{c1}(i) \vee t_{c2}(i)\}$, $m = \sum_{i=0}^{n} \{t_{c1}(i) \wedge t_{c2}(i)\}$

$Pr(c1 \equiv c2)$ represents the probability of the instants $c1$ that are coincident with the instants of $c2$. `Coincidence` relation is bidirectional, which means that $c1$ and $c2$ are equivalent in the relation. In this case, k is the total number of steps at which either $c1$ or $c2$ ticks. m is the number of ticks of steps at which `coincidence` relation is satisfied, i.e., the steps at which both $c1$ and $c2$ tick.

The inverse of `coincidence` relation, called `exclusion`, hinders two clocks from ticking simultaneously.

Definition 5 (Probabilistic Exclusion). *The probability of exclusion relation between* $c1$ *and* $c2$*, denoted* $c1 \#_p c2$*, is satisfied if the following conditions hold:*

$$\mathcal{M} \vDash c1 \#_p c2 \iff Pr(c1 \# c2) \geq p, \quad where \quad Pr(c1 \# c2) = \frac{m}{k},$$

$$k = \sum_{i=0}^{n} \{t_{c1}(i) \vee t_{c2}(i)\},$$

$$m = \sum_{i=0}^{n} \{(t_{c1}(i) \wedge \neg t_{c2}(i)) \vee (\neg t_{c1}(i) \wedge t_{c2}(i))\}$$

k is the total number of steps at which either $c1$ or $c2$ ticks. m indicates the number of steps at which `exclusion` relation is satisfied, i.e., the steps at which only one of the two clocks ticks.

The probabilistic extension of `causality` and `precedence` relations are defined based on the history of the clocks. Recall that $h_{c1}(i)$ ($h_{c2}(i)$) indicates how many times $c1$ ($c2$) has ticked before the step i. If the history of $c1$ is greater than the one of $c2$ at the same step, we say that $c1$ runs faster than $c2$ at that step. `Causality` relation specifies that an event causes another one, i.e., the effect cannot occur if the cause has not.

Definition 6 (Probabilistic Causality). *The probabilistic causality relation between* $c1$ *and* $c2$ *(*$c1$ *is the cause and* $c2$ *is the effect), denoted,* $c1 \preceq_p c2$*, is satisfied if the following conditions hold:*

$$\mathcal{M} \vDash c1 \preceq_p c2 \iff Pr(c1 \preceq c2) \geq p$$

where $Pr(c1 \preceq c2) = \frac{m}{k}$, $k = \sum_{i=0}^{n} t_{c1}(i)$, $m = \sum_{i=0}^{n} \{t_{c1}(i) \wedge h_{c1}(i) \geq h_{c2}(i)\}$

k is the total number of ticks of $c1$. m is the number of ticks of $c1$ satisfying `causality` relation. A tick of $c1$ satisfies `causality` relation if $c2$ does not occur prior to $c1$, i.e., the history of $c2$ is less than or equal to the history of $c1$ at the current step.

The strict `causality`, called `precedence`, constrains that one clock must always run faster than the other.

Definition 7 (Probabilistic Precedence). *The probabilistic precedence relation between* $c1$ *and* $c2$*, denoted,* $c1 \prec_p c2$*, is satisfied if the following conditions hold:*

$$\mathcal{M} \models c1 \prec_p c2 \iff Pr(c1 \prec c2) \geq p, \quad \textit{where}$$

$$Pr(c1 \prec c2) = \frac{m}{k}, \quad k = \sum_{i=0}^{n} t_{c1}(i),$$

$$m = \sum_{i=0}^{n} \underbrace{t_{c1}(i) \wedge h_{c1}(i) \geq h_{c2}(i)}_{\textit{(1)}} \wedge \underbrace{(h_{c1}(i) = h_{c2}(i) \implies \neg t_{c2}(i))}_{\textit{(2)}}$$

k indicates the total number of ticks of $c1$. m is the number of ticks of $c1$ satisfying `precedence` and holding the two conditions: (1) the history of $c1$ is greater than or equal to the history of $c2$ at the same step; (2) $c1$ and $c2$ must not be coincident, i.e., when the history of $c1$ and $c2$ are equal, $c2$ must not tick.

5 Translation of CCSL and PrCCSL into SDV

In order to formally prove the EAST-ADL timing constraints (given in Sect. 3) using SIMULINK DESIGN VERIFIER (SDV), we investigate how those constraints, specified in CCSL *expressions* and PrCCSL *relations* (`Spec. R.ID` in Fig. 2), can be translated into *Proof Objective Models* (POM). CCSL *expressions* construct new clocks and the *relations* between the new clocks are specified using PrCCSL. We first provide strategies that represent CCSL *expressions* in SIMULINK/STATEFLOW (S/S). We then present how the EAST-ADL timing constraints defined in PrCCSL can be translated into the corresponding POMs, which are integrated with the S/S models of CCSL *expressions*, based on the strategies.

5.1 Mapping CCSL Expressions into S/S

We first describe how tick and history of CCSL can be mapped to corresponding S/S models. Using the mapping, we show CCSL *expressions* can be modeled in S/S. A "step" (defined in Sect. 2) is represented as a sample time in SIMULINK and set to 0.001 s. The clock ticks are expressed as boolean variables (1 "ticking" or 0 "non-ticking") during simulation. The history of clock (expressed as integer) is increased as the clock ticks and is interpreted as a function `His(c)` in Fig. 4: Since h_c, the history of clock c, is determined by the value of c at the immediate precedent step, a ≪Delay≫ block is employed to delay c by one step. Whenever c ticks at the prior step, ≪ES≫ is executed and increases h_c by 1.

Based on the mapping patterns of tick and history, we present how `PeriodicOn`, `DelayFor`, `Infimum` and `Supremum` *expressions* can be represented as S/S models.

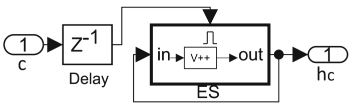
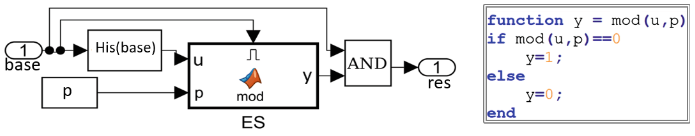


Fig. 4. h_c = `His(c)`

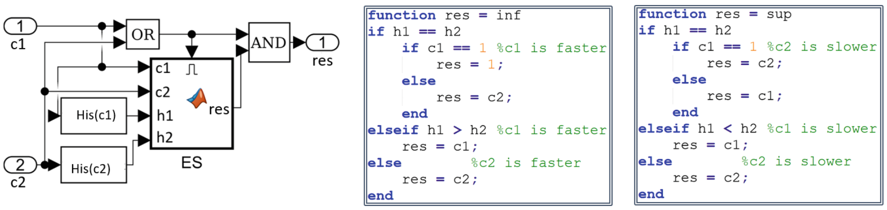
PeriodicOn: $res \triangleq$ `PeriodicOn` $base$ `period` p, where $\triangleq$ means "is defined as", builds a new clock res based on $base$ clock and a $period$ parameter p, i.e., res ticks at every p^{th} tick of $base$. The SIMULINK model of `PeriodicOn` is illustrated in Fig. 5: When $base$ ticks, the ≪Matlab Function≫ (code is shown in the box), embedded in the ≪ES≫ subsystem, is triggered and checks if the history of the $base$, `His(base)`, is an integral multiple of p. When $base$ ticks and its history is equal to the integral multiple of p, res ticks. The `PeriodicOn` S/S model is employed for the translation of EAST-ADL `Periodic` timing constraint (R1 in Fig. 2) into its POM in SDV.



```
function y = mod(u,p)
if mod(u,p)==0
    y=1;
else
    y=0;
end
```

Fig. 5. $res \triangleq$ `PeriodicOn` $base$ `period` p

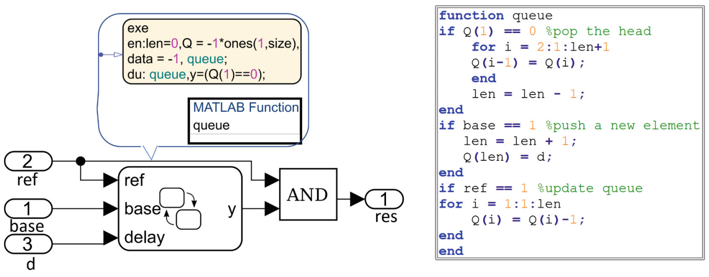
Infimum (resp. Supremum): $res \triangleq$ `Inf`($c1$, $c2$) (resp. `Sup`($c1$, $c2$)), creates a new clock res, which is the slowest (resp. fastest) clock faster (resp. slower) than the two clocks, $c1$ and $c2$. In other words, res ticks at the step whereby the faster (slower) clock between $c1$ and $c2$ ticks. The SIMULINK model of `Infimum` (resp. `Supremum`) is depicted in Fig. 6. When $c1$ or $c2$ ticks, the `inf` (resp. `sup`) function embedded in ≪ES≫ is executed and decides which clock is faster (resp. slower) than the other by comparing the history of $c1$ and $c2$ (h1 and h2). If the clock (either $c1$ or $c2$) ticking at the current step is the faster (resp. slower) clock, res ticks. The `Infimum` and `Supremum` S/S models are utilized for the translation of EAST-ADL `Synchronization` timing constraint (R5 in Sect. 3) into POM.



```
function res = inf
if h1 == h2
    if c1 == 1 %c1 is faster
        res = 1;
    else
        res = c2;
    end
elseif h1 > h2 %c1 is faster
    res = c1;
else           %c2 is faster
    res = c2;
end
```

```
function res = sup
if h1 == h2
    if c1 == 1 %c2 is slower
        res = c2;
    else
        res = c1;
    end
elseif h1 < h2 %c1 is slower
    res = c1;
else           %c2 is slower
    res = c2;
end
```

Fig. 6. $res \triangleq$ `Inf`($c1$, $c2$) (rep. `Sup`($c1$, $c2$))



```
function queue
if Q(1) == 0 %pop the head
    for i = 2:1:len+1
    Q(i-1) = Q(i);
    end
    len = len - 1;
end
if base == 1 %push a new element
   len = len + 1;
   Q(len) = d;
end
if ref == 1 %update queue
for i = 1:1:len
    Q(i) = Q(i)-1;
end
end
```

Fig. 7. $res \triangleq base$ `DelayFor` d `on` ref

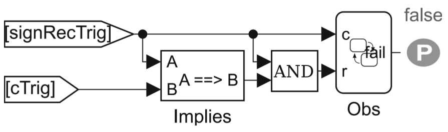
DelayFor: $res \triangleq base$ `DelayFor` d `on` ref, constructs a new clock res based on $base$ clock and *reference* clock (ref), i.e., each time $base$ ticks, res ticks at the d^{th} tick of ref. The SIMULINK model of `DelayFor` is shown in Fig. 7: A STATEFLOW chart is utilized to observe the ticks of $base$ and ref. A queue, Q, and its enqueue/dequeue operation is implemented in the function *queue*. y indicates whether ref has ticked d times since $base$ ticked. When $base$ ticks ($base == 1$), an element with value d is enqueued, and each time ref ticks, the value of the element is decreased by 1. After d ticks of ref, the element becomes 0 and y becomes true. An ≪And≫ block is applied to delimit that the tick of res must coincide with the tick of ref (i.e., res is a `subclock` of ref). The `DelayFor` S/S model is adapted to construct the POM models of EAST-ADL timing requirements R3–R7 in Sect. 3.

5.2 Representation of PrCCSL in SDV

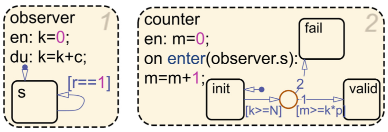
We present how the translation of EAST-ADL timing constraints (specified in PrCCSL *relations* and CCSL *expressions*) can be interpreted as POMs in the view point of analysis engine SDV. Recall the definitions of PrCCSL in Sect. 4. A PrCCSL *relation* is valid if the probability of a *relation* ϕ being satisfied is greater than or equal to the given probability threshold p. It can be interpreted as *Hypothesis Testing* [30]: Decide whether $\mathcal{M} \models \Pr(\phi) \geq$ p (hypothesis H_0) against $\mathcal{M} \models \Pr(\phi) <$ p (alternative hypothesis H_1).

Probabilistic Subclock is employed to specify EAST-ADL `Periodic` timing constraint, given as $signRecTrig \subseteq_p cTrig$ (Spec. R2 in Fig. 2). The corresponding POM is shown in Fig. 8: The STATEFLOW chart *Obs* in Fig. 8(b) is utilized for *Hypothesis Testing*, where k is the total number of ticks of $signRecTrig$ (`subclock`) and m is the number of ticks satisfying the `subclock` *relation*.

Whenever $signRecTrig$ ticks, k is increased by 1, and if the `subclock` *relation* holds on that tick (i.e., the condition "$signRecTrig \implies cTrig$" is true), m is increased by 1. When k is increased to the sample size N, the STATEFLOW chart then judges whether the number of "success" ticks of $signRecTrig$ is greater than or equal to "$p * k$" (i.e., whether $\frac{m}{k} \geq p$ is valid), and it activates either *valid* ("H_0" is accepted) or *fail* state ("H_1" is accepted). A ≪Proof Objective≫



(a) $signRecTrig \subseteq_p cTrig$

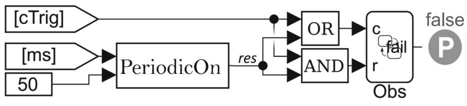


(b) *Obs* Chart

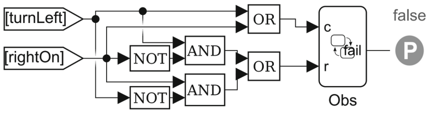
Fig. 8. POM of Probabilistic `Subclock`

block with *false* value is employed to check whether the probabilistic `subclock` *relation* is satisfied, i.e., *fail* is never reached. Similarly, using the *Obs* chart, other PrCcsl *relations* can be represented as POMs. Further details are given below.

Probabilistic Coincidence is adapted to specify East-ADL `Periodic` timing constraint, given as $cTrig \equiv_p$ {`PeriodicOn` *ms* `period` 50} (Spec. R1 in Fig. 2). The representative POM is shown in Fig. 9(a): A `PeriodicOn` subsystem (whose internal blocks are shown in Fig. 5) is utilized to generate a periodic clock *res* that ticks every 50 ms. According to Definition 4 in Sect. 4, if either *cTrig* or *res* ticks ("*cTrig* `OR` *res*" is true), *c* becomes true and *k* is increased by 1. Meanwhile, if *cTrig* and *res* tick simultaneously ("*cTrig* `AND` *res*" is true), *r* becomes true and *m* is increased by 1. Based on the value of *m* and *k*, *Obs* checks whether the probability of `coincidence` *relation* being satisfied is greater than or equal to *p* and activates either *valid* or *fail* state. ≪Proof Objective≫ block checks whether *fail* state is always inactive, i.e., H_0 is accepted.



(a) $cTrig \equiv_p$ {`PeriodicOn` *ms* `period` 50}



(b) $turnLeft \ \#_p \ rightOn$

Fig. 9. POM of Probabilistic `Coincidence` and `Exclusion`

Probabilistic Exclusion is utilized to specify East-ADL `Exclusion` timing constraint, given as $turnLeft \ \#_p \ rightOn$ (Spec. R8 in Fig. 2). The corresponding POM is shown in Fig. 9(b): *k* is increased by 1 when either *turnLeft* or *rightOn* ticks. If only one of the two clocks ticks at the current step, i.e., *r* (the input of *Obs*) is true, *m* is increased by 1. ≪Proof Objective≫ block with false value checks whether *fail* state is never reached, i.e., H_0 is accepted.

Probabilistic Causality is employed to specify East-ADL `Synchronization` timing constraint, $sup \preceq_p$ {*inf* `DelayFor` 40 `on` *ms*} (Spec. R5 in Fig. 2), where *sup* (*inf*) is the fastest (slowest) event slower (faster) than the five input events, *speed*, *signType*, *direct*, *gear* and *torque*. *sup* and *inf* are defined as:

$$sup \triangleq \texttt{Sup}(\texttt{Sup}(speed,\ signType),\ \texttt{Sup}(\texttt{Sup}(direct,\ gear),\ torque)) \quad (1)$$

$$inf \triangleq \texttt{Inf}(\texttt{Inf}(speed,\ signType),\ \texttt{Inf}(\texttt{Inf}(direct,\ gear),\ torque)) \quad (2)$$

The representative POM is illustrated in Fig. 10: The S/S models of `Inf` and `Sup` (shown in Fig. 6) are utilized in order to construct *inf* (1) and *sup* (2), modeled as `INF` and `SUP` subsystems, respectively. A new clock *dinf* is generated by delaying *inf* for 40 ticks of *ms*, i.e., $dinf \triangleq \{inf$ `DelayFor` 40 `on` $ms\}$, and it is represented by using the S/S model of `DelayFor` (shown in Fig. 7). Then `Probabilistic Causality` *relation* between *sup* and *dinf* is checked. According to Definition 6, when *sup* ticks, *k* is increased by 1. At the same step, if the `causality` *relation* between *sup* and *dinf* is satisfied, i.e., the history of *sup* is greater than or equal to the history of *dinf*, *m* is increased by 1. ≪Proof Objective≫ block analyzes if the `Probabilistic Causality` *relation* is satisfied, i.e., the *fail* state is never activated. Similarly, EAST-ADL `Execution` (R3) and `Comparison` (R7) timing constraints specified in `Probabilistic Causality` using `DelayFor` can be translated into corresponding POMs. For further details, refer to [17].

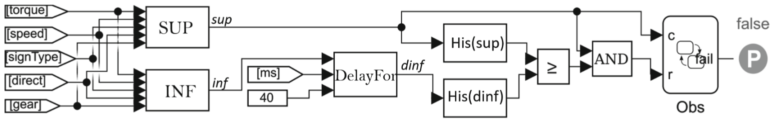


Fig. 10. $sup \preceq_p \{inf$ `DelayFor` 40 `on` $ms\}$

Probabilistic Precedence is used to specify EAST-ADL `Sporadic` timing constraint, given as $\{obstc$ `DelayFor` 500 `on` $ms\} \prec_p veRun$ (Spec. R6 in Fig. 2). The constraint delimits that two events *obstc* and *veRun* must have a minimum delay 500 ms, and its corresponding POM is illustrated in Fig. 11: A new clock *res* is generated by delaying *obstc* by 500 ticks of *ms*, i.e., $res \triangleq \{obstc$ `DelayFor` 500 `on` $ms\}$, and it is modeled using the S/S model of `DelayFor`. Then R6 can be checked by verifying $res \prec_p veRun$. As presented in Fig. 11, whenever *res* ticks, *c* becomes true and *k* is increased by 1. If the tick of *obstc* satisfies the `precedence` *relation*, i.e., the history of *res* is greater than or equal to the history of *veRun* (excludes *res* and *veRun* are coincident), *r* becomes true and *m* is increased by 1. ≪Proof Objective≫ block checks whether `Probabilistic Precedence` is

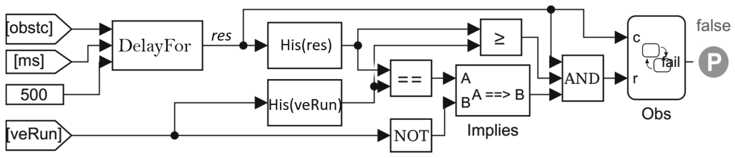


Fig. 11. $\{obstc$ `DelayFor` 500 `on` $ms\} \prec_p veRun$

satisfied, i.e., the *fail* state is never activated. Similarly, EAST-ADL `End-to-End` timing constraint (R4) specified in `Probabilistic Precedence` can be translated into its corresponding POM [17].

6 Experiments: Verification and Validation

We have presented how the EAST-ADL timing constraints, specified in PrCCSL *relations* and CCSL *expressions* are converted to POMs. To enable verification of the timed and stochastic behaviors of AV using SDV, the behavior of each f_p is described in S/S. The $\texttt{FA}_{SYS}$, consisting of a set of S/S is considered the entire behavior model of AV. To describe the stochastic environments of AV, a pseudo random number generator, *Mersenne Twister* [26] implemented in MATLAB script is employed: 1. The traffic signs (6 types) are randomly recognized by AV and the probability of each sign type occurring is equally set as 16.7%; 2. The probability of AV being obstructed by any obstacles is set to maximum 5%; 3. Since AV runs under different road conditions, speed variation influenced by the conditions ranges within [0, 2] m/s. We have formally specified and analyzed over 30 properties (associated with timing constraints) of the AV system [17]. A list of selected properties (Sect. 3) are verified using SDV and the results are listed in Table 1. The simulation bound and the probability threshold are set to 60000 steps and 95% respectively. A maximum of 4 properties per EAST-ADL timing constraint are verified and all properties are established as valid. For further details regarding the full POMs and S/S models used in the experiment, refer to [7,17].

Table 1. Consolidated Verification Results in SDV

Req	Category	Expression	Result	Time (Min)	Mem (Mb)	CPU (%)
R1	Periodic	$cTrig \equiv_{\mathbf{0.95}} \{\mathbf{PeriodicOn}\ ms\ \mathbf{period}\ 50\}$	valid	22.10	1303	11.90
R2	Periodic	$signRTrig \subseteq_{\mathbf{0.95}} cTrig$	valid	60.15	1455	11.24
R3	Execution	$\{imIn\ \mathbf{DelayFor}\ 100\ \mathbf{on}\ ms\} \preceq_{\mathbf{0.95}} signOut$	valid	38.20	1319	9.94
		$signOut \preceq_{\mathbf{0.95}} \{imIn\ \mathbf{DelayFor}\ 150\ \mathbf{on}\ ms\}$	valid	33.96	879	9.66
R4	End-to-End	$\{signIn\ \mathbf{DelayFor}\ 150\ \mathbf{on}\ ms\} \prec_{\mathbf{0.95}} tqOut$	valid	35.95	1334	11.69
		$tqOut \prec_{\mathbf{0.95}} \{signIn\ \mathbf{DelayFor}\ 250\ \mathbf{on}\ ms\}$	valid	24.95	1219	14.24
R5	Synchronization	$sup \preceq_{\mathbf{0.95}} \{inf\ \mathbf{DelayFor}\ 40\ \mathbf{on} ms\}$	valid	38.95	1282	12.15
R6	Sporadic	$\{obstc\ \mathbf{DelayFor}\ 500\ \mathbf{on}\ ms\} \prec_{\mathbf{0.95}} veRun$	valid	100.5	1212	12.79
R7	Comparison	$\{signIn\ \mathbf{DelayFor}\ 250\ \mathbf{on}\ ms\} \preceq_{\mathbf{0.95}}$ $\{signIn\ \mathbf{DelayFor}\ (Wctrl + Wvd)\ \mathbf{on}\ ms\}$	valid	17.88	1050	6.87
R8	Exclusion	$turnLeft\ \#_{\mathbf{0.95}}\ rightOn$	valid	387.76	1139	8.25

7 Related Work

Considerable research efforts have been devoted to formal analysis of CPS by applying SDV [12,14], which are however, limited to the functional properties without consideration of non-functional properties, i.e., timing constraints.

In the context of East-adl, efforts on the integration of East-adl and formal techniques based on timing constraints were investigated in several works [13,16,23,29], which are however, restricted to the executional aspects of system functions without addressing stochastic behaviors. Kang [22] and Suryadevara [32,33] defined the execution semantics of both the controller and the environment of industrial systems in Ccsl which are given as mapping to Uppaal models amenable to model checking. In contrast to our current work, those approaches lack precise probabilistic annotations specifying stochastic properties. Zhang [34] transformed Ccsl into first order logics that are verifiable using SMT solver. However, this work is limited to functional properties, and no timing constraints are addressed. Though, Kang et al. [15,19] and Marinescu et al. [25] presented both simulation and model checking approaches of Simulink and Uppaal-SMC on East-adl models, neither formal specification nor verification of extended East-adl timing constraints with probability were conducted. Our approach is a first application on the integration of East-adl and formal V&V techniques based on probabilistic extension of East-adl/Tadl2 constraints using SDV. An earlier study [18,20,21] defined a probabilistic extension of East-adl timing constraints and presented model checking approaches on East-adl models, which inspires our current work. Specifically, the techniques provided in this paper define new operators of Ccsl with stochastic extensions (PrCcsl) and formally verify the extended East-adl timing constraints of CPS. Du et al. [11] proposed the use of Ccsl with probabilistic logical clocks to enable stochastic analysis of hybrid systems by limiting the possible solutions of clock ticks. Whereas, our work is based on the probabilistic extension of East-adl timing constraints with the focus on probabilistic verification of the extended constraints, particularly, in the context of WH.

8 Conclusion

We present an approach to perform probabilistic analysis of East-adl timing constraints in automotive systems at the early design phase: 1. Probabilistic extension of Ccsl, called PrCcsl, is defined and the East-adl/Tadl2 timing constraints with stochastic properties are specified in PrCcsl; 2. The semantics of the extended constraints in PrCcsl, captured in Simulink/Stateflow, is translated into verifiable POMs for formal verification; 3. A set of mapping rules is proposed to facilitate guarantee of translation. Our approach is demonstrated on an autonomous traffic sign recognition vehicle (AV) case study. Although, we have shown that defining and translating a subset of Ccsl with probabilistic extension into POMs is sufficient to verify East-adl timing constraints, as ongoing work, advanced techniques covering a full set of Ccsl constraints are further studied. Despite the fact that SDV supports probabilistic analysis of the timing constraints of AV, the computational cost of verification in terms of time is rather expensive. Thus, we continuously investigate complexity-reducing design/mapping patterns for CPS to improve effectiveness and scalability of system design and verification.

Acknowledgment. This work is supported by the NSFC 46000-41030005.

References

1. Automotive open system architecture. https://www.autosar.org/
2. Simulink Design Verifier. https://www.mathworks.com/help/sldv
3. IEC 61508: Functional safety of electrical electronic programmable electronic safety related systems. International Organization for Standardization, Geneva (2010)
4. EAST-ADL specification v2.1.9. Technical report, MAENAD (2011). https://www.maenad.eu/public/EAST-ADL-Specification_M2.1.9.1.pdf
5. ISO 26262–6: Road vehicles functional safety part 6. Product development at the software level. International Organization for Standardization, Geneva (2011)
6. MAENAD (2011). http://www.maenad.eu/
7. Simulink library of PrCCSL (2018). https://github.com/huangl223/PrCCSL
8. André, C.: Syntax and semantics of the clock constraint specification language (CCSL). Ph.D. thesis, INRIA (2009)
9. Bernat, G., Burns, A., Llamosi, A.: Weakly hard real-time systems. Trans. Comput. **50**(4), 308–321 (2001)
10. Blom, H., et al.: TIMMO-2-USE timing model, tools, algorithms, languages, methodology, use cases. Technical report, TIMMO-2-USE (2012)
11. Du, D., Huang, P., Jiang, K., Mallet, F., Yang, M.: MARTE/pCCSL: modeling and refining stochastic behaviors of CPSs with probabilistic logical clocks. In: Kouchnarenko, O., Khosravi, R. (eds.) FACS 2016. LNCS, vol. 10231, pp. 111–133. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57666-4_8
12. Gholami, M.R.: Verifying timed LTL properties using Simulink Design Verifier. Ph.D. thesis, École Polytechnique de Montréal (2016)
13. Goknil, A., Suryadevara, J., Peraldi-Frati, M.-A., Mallet, F.: Analysis support for TADL2 timing constraints on EAST-ADL models. In: Drira, K. (ed.) ECSA 2013. LNCS, vol. 7957, pp. 89–105. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39031-9_8
14. Etienne, J.-F., Fechter, S., Juppeaux, E.: Using simulink design verifier for proving behavioral properties on a complex safety critical system in the ground transportation domain. Sci. Comput. Program. **77**(10), 1151–1177 (2010)
15. Kang, E.Y., Chen, J., Ke, L., Chen, S.: Statistical analysis of energy-aware real-time automotive systems in EAST-ADL/Stateflow. In: ICIEA, pp. 1328–1333. IEEE (2016)
16. Kang, E.Y., Enoiu, E.P., Marinescu, R., Seceleanu, C., Schobbens, P.Y., Pettersson, P.: A methodology for formal analysis and verification of EAST-ADL models. Reliab. Eng. Syst. Saf. **120**(12), 127–138 (2013)
17. Kang, E.Y., Huang, L.: Formal specification & analysis of autonomous systems in PrCCSL/Simulink Design Verifier. Technical report, SYSU (2018). https://sites.google.com/site/kangeu/home/publications
18. Kang, E.Y., Huang, L., Mu, D.: Formal verification of energy and timed requirements for a cooperative automotive system. In: SAC, pp. 1492–1499. ACM (2018)
19. Kang, E.Y., Ke, L., Hua, M.Z., Wang, Y.X.: Verifying automotive systems in EAST-ADL/Stateflow using UPPAAL. In: APSEC, pp. 143–150. IEEE (2015)
20. Kang, E.Y., Mu, D., Huang, L., Lan, Q.: Model-based analysis of timing and energy constraints in an autonomous vehicle system. In: QRS, pp. 525–532. IEEE (2017)
21. Kang, E.Y., Mu, D., Huang, L., Lan, Q.: Verification and validation of a cyber-physical system in the automotive domain. In: QRS, pp. 326–333. IEEE (2017)

22. Kang, E.Y., Schobbens, P.Y.: Schedulability analysis support for automotive systems: from requirement to implementation. In: SAC, pp. 1080–1085. ACM (2014)
23. Kang, E.-Y., Schobbens, P.-Y., Pettersson, P.: Verifying functional behaviors of automotive products in EAST-ADL2 using UPPAAL-PORT. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 243–256. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24270-0_18
24. Mallet, F., De Simone, R.: Correctness issues on MARTE/CCSL constraints. Sci. Comput. Program. **106**, 78–92 (2015)
25. Marinescu, R., Kaijser, H., Mikučionis, M., Seceleanu, C., Lönn, H., David, A.: Analyzing industrial architectural models by simulation and model-checking. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2014. CCIS, vol. 476, pp. 189–205. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17581-2_13
26. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. TOMACS **8**(1), 3–30 (1998)
27. Nicolau, G.B.: Specification and analysis of weakly hard real-time systems. Trans. Comput. pp. 308–321 (1988)
28. Object Management Group: UML profile for MARTE: modeling and analysis of real-time embedded systems. Technical report (2011)
29. Qureshi, T.N., Chen, D.J., Persson, M., Törngren, M.: Towards the integration of UPPAAL for formal verification of EAST-ADL timing constraint specification. In: TiMoBD workshop (2011)
30. Reijsbergen, D., Boer, P.T.D., Scheinhardt, W., Haverkort, B.: On hypothesis testing for statistical model checking. STTT **17**(4), 377–395 (2015)
31. Simulink and Stateflow. https://www.mathworks.com/products.html
32. Suryadevara, J.: Validating EAST-ADL timing constraints using UPPAAL. In: SEAA, pp. 268–275. IEEE (2013)
33. Suryadevara, J., Seceleanu, C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL mode behaviors using UPPAAL. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 1–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_1
34. Zhang, M., Ying, Y.: Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems. ACM SIGPLAN Not. **52**(4), 61–70 (2017)

Mixed-Criticality Scheduling with Limited HI-Criticality Behaviors

Zhishan Guo[1,2(✉)], Luca Santinelli[3], and Kecheng Yang[4,5]

[1] University of Central Florida, Orlando, USA
[2] Missouri University of Science and Technology, Rolla, USA
guozh@mst.edu
[3] ONERA, Toulouse, France
luca.santinelli@onera.fr
[4] Texas State University, San Marcos, USA
[5] University of North Carolina at Chapel Hill, Chapel Hill, USA
yangk@cs.unc.edu

Abstract. Due to size, weight, and power considerations, there is an emerging trend in real-time embedded systems design towards implementing functionalities of different levels of importance upon a shared platform, or implementing Mixed-Criticality (MC) systems. Much existing work on MC scheduling focuses on the classic Vestal model, where upon a mode switch, it is pessimistically assumed that all tasks may simultaneously exceed their less pessimistic execution time estimations, or LO-WCETs. In this paper, a less pessimistic MC model is proposed for system designers to specify the maximum number of tasks that may simultaneously exceed their LO-WCETs. The applicability and schedulability of the classic EDF-VD scheduler under this newly proposed model are studied, and a new schedulability test is presented. Experiments demonstrate that, by applying the proposed model and new schedulability test, significantly better schedulability can be achieved.

1 Introduction and Motivation

The Worst-Case Execution Time (WCET) abstraction models the execution behavior of real-time tasks. Given a piece of code to execute upon a specified platform, the WCET is an upper bound to the time duration needed to finish the execution of a single invocation of that piece of code. Unfortunately, even when severe restrictions are placed upon the structure of the code e.g., known loop bounds, it is still difficult to determine the exact WCET. Furthermore, the occurrence of the WCET is usually extremely unlikely, unless under highly pathological circumstances such as faults.

In order to utilize the significant gap between the actual running time and the WCET, it has been proposed to implement functionalities of different degrees

Supported by NSF grants CNS 1850851, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and fundings from General Motors and Center for Advancing Faculty Excellence of the University of Missouri System.

X. Feng et al. (Eds.): SETTA 2018, LNCS 10998, pp. 187–199, 2018.
https://doi.org/10.1007/978-3-319-99933-3_13

of importance (or criticalities) upon a shared platform. Under such design, for each of the more important tasks, a less pessimistic execution time estimation is also provisioned in addition to the most pessimistic WCET. When the more important tasks actually complete by these less pessimistic estimations, less important tasks are allowed to execute as well, so that processor capacities are not wasted. In contrast, in occasional situations where the more important tasks execute beyond their less pessimistic estimations, the less important tasks may be dropped. In order to validate systems under this design approach, Mixed-Criticality (MC) scheduling techniques are needed.

Prior research on MC scheduling (see [7] for an up-to-date review) focused on the Vestal model [14], which assigns multiple WCET estimations for each individual task. Typically, in the two-criticality-level case, each task is designated as being of either higher (HI) or lower (LO) criticality. Two WCETs are specified for each HI-criticality task: a LO-WCET and a larger HI-WCET which could be larger than the LO-WCET by several orders of magnitude. One WCET is specified for each LO-criticality task: the LO-WCET.

The Vestal model defines two system *modes*, each associated with different guarantees. In the normal mode, every HI-criticality task completes its execution by its LO-WCET, and each LO-criticality task should be guaranteed to execute up to its LO-WCET as well. On the other hand, whenever any HI-criticality task does not signal its completion after exhausting its LO-WCET, a system mode switch will be triggered; in the new mode, all of the LO-criticality tasks are dropped in order to guarantee every HI-criticality task to execute up to its HI-WCET. In this traditional MC Scheduling model, *all* HI-criticality tasks may *simultaneously* exceed their LO-WCETs, requiring executions up to their HI-WCETs in the new mode.

Motivation. However, in some cases, this assumption about the execution of HI-criticality tasks in the classic Vestal model could be too pessimistic. Indeed, having all HI-criticality tasks simultaneously exceeding their LO-WCETs could be non-representative of many real-world real-time embedded systems.

```
task  Anti_Frozen {
   f1();
   t = temperature();
   if (t < 0) {
     f2();
   }
   if (t < -20) {
     f3();
   }
}
```

```
task  Over_Heat {
   g1();
   t = temperature();
   if (t > 50) {
     g2();
   }
   if (t > 80) {
     g3();
   }
}
```

Fig. 1. Two example tasks in a safety critical system.

The following two pieces of codes shown in Fig. 1 is a toy example to illustrate this motivation in more details.

Let us assume both tasks are HI-criticality tasks as they perform some important safety features of the system, dealing with either frozen (task Anti_Frozen) or over-heating (task Over_Heat) situation. Under normal circumstances, actions in f2() and g2() are more than enough to bring the ambient temperature (around the platform) back to normal range (0 to 50), such that f3() and g3() will not need to be executed. As a result, we may assign LO-WCET of task Anti_Frozen as the maximum time to execute f1() and f2(); HI-WCET of task Anti_Frozen as the maximum time to execute f1(), f2(), and f3(); LO-WCET of task Over_Heat as the maximum time to execute g1() and g2(); and HI-WCET of task Over_Heat as the maximum time to execute g1(), g2(), and g3().

A straightforward observation is that, even under extreme situations, only *one* of the above two tasks will need to execute their final *if* branches; i.e., there will not be any time instant that both tasks require executions of their HI-WCETs simultaneously. As a result, any analysis following the Vestal model is over pessimistic in this example, as it will need to take the impossible case into consideration, where both tasks exceed their LO-WCETs at the same time.

Note that, when all HI-criticality tasks simultaneously exceed their LO-WCET due to certain system degradation or failure, it is computationally more efficient to characterize such behaviors with the MC varying speed model [5,6,8–10], which better represents the uncertainties arising from the executing speed of the platform, rather than with Vestal model by using multiple estimations of WCETs (due to the NP-hardness [1]).

Contribution. In this work, we propose a new MC system model to cope with more realistic assumptions for real-time embedded systems. The proposed model is more general than the existing well-studied Vestal model in the sense that it allows a system designer to specify the number of HI-criticality tasks that can exceed their LO-WCET simultaneously. We then analyze how this additional specification could impact the schedulability and develop an MC scheduler for this new model. We finally conduct schedulability experiments and compare the results from our scheduler and a classic MC scheduler, namely EDF-VD. The advantages from having only subsets of HI-criticality tasks exceeding their LO-WCET thresholds simultaneously are validated by these experimental results.

Organization. Section 2 describes the proposed MC system model. Section 3 adapts an existing scheduler for the problem, and proves its correctness. Section 4 evaluates the performance of the proposed scheduler under various parameter settings, and compares it with an existing MC task scheduler. Section 5 concludes the work and points out some future directions.

2 Model and Definitions

Mixed-Criticality Tasks. A MC periodic task set τ is specified as a finite collection of MC periodic tasks, each of which generates an unbounded number of MC jobs. Each task τ_i has a period, T_i, modeling the time separation between two consecutive jobs of this task, and each job of τ_i has to complete its execution by D_i time units. In this paper, the tasks are assumed to have implicit deadlines, i.e., $D_i = T_i$. The integer time model is also assumed—all task periods are non-negative integers and all job arrivals occur at integer time instants.

We consider a uniprocessor system where all tasks execute on and share the single processor, while the scheduler determines how it is shared.

A task exhibits LO-criticality behavior if *all* of its jobs complete execution by its LO-WCET. In contrast, a task is in HI-criticality behavior if *any* of its jobs requires an execution longer than its LO-WCET, but no more than its HI-WCET. Any other behavior is considered *erroneous*.

A HI-criticality task τ_i can be specified by $\tau_i = ([c_i(\text{LO}), c_i(\text{HI})], T_i)$. T_i is the period and the relative deadline of task τ_i; $[c_i(\text{LO}), c_i(\text{HI})]$ is the tuple of WCET estimations, $c_i(\text{LO})$ for the LO-WCET and $c_i(\text{HI})$ for the HI-WCET, where $c_i(\text{LO}) \leq c_i(\text{HI})$.

A LO-criticality task τ_j is represented with two parameters $\tau_j = (c_j(\text{LO}), T_j)$. T_j is the period and the deadline of the task and $c_j(\text{LO})$ characterizes the LO-criticality mode worst-case execution time. For LO-criticality tasks only the LO-criticality behavior is possible.

The two WCETs specified for each HI-criticality task τ_i may come from timing analysis tools with different levels of pessimism:

- $c_i(\text{LO})$, which is determined by a less pessimistic timing analysis tool (or with less guarantees of being the worst-case for any possible execution condition) — a HI-criticality task may require an execution length of more than $c(\text{LO})$; and
- $c_i(\text{HI})$, which is sometimes larger than the LO-WCET by several orders of magnitude — it may be determined by a more conservative timing analysis, and it presents the worst-case execution time for any possible execution condition the task may experience.

The *utilizations* of tasks are defined for HI- and LO-criticality tasks respectively. Each HI-criticality task has two associated utilizations—one in each mode, whereas each LO-criticality task has only one associated utilization as follows:

- $U_{\text{HI}}^{\text{HI}}(\tau_i) = c_i(\text{HI})/T_i$ - HI-criticality task utilization in HI-criticality mode;
- $U_{\text{HI}}^{\text{LO}}(\tau_i) = c_i(\text{LO})/T_i$ - HI-criticality task utilization in LO-criticality mode;
- $U_{\text{LO}}^{\text{LO}}(\tau_i) = c_i(\text{LO})/T_i$ - LO-criticality utilization.

Mixed-Criticality Systems. An MC system is defined to run under two possible *modes*: a normal mode (LO-criticality mode) where every job completes upon executing for no more than its LO-WCET and a HI-criticality mode where some HI-criticality job executes for more than its LO-WCET but imperatively completes upon execution for no more than its HI-WCET.

The system mode will be switched from LO-criticality mode to HI-criticality mode if *any* HI-criticality task has exhausted its LO-WCET but has not completed. Only HI-criticality tasks are guaranteed to be met their deadlines under HI-criticality mode.

Definition 1 (MC Task Instance). *A MC task instance I is composed of an MC task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$, where both* HI*-criticality tasks and* LO*-criticality tasks may be in τ. n_{HI} denotes the number of* HI*-criticality tasks in τ, and $n_{\text{HI}} \leq n$. Each* HI*-criticality task τ_i is represented as $\tau_i = ([c_i(\text{LO}), c_i(\text{HI})], T_i)$, while each* LO*-criticality tasks τ_j is represented as $\tau_j = (c_j(\text{LO}), T_j)$.*

The notion of *utilization difference* for HI-criticality tasks is defined as follows.

Definition 2 (Utilization Difference). *The utilization difference of a* HI*-criticality task τ_i is defined by*

$$\delta_i = \frac{c_i(\text{HI}) - c_i(\text{LO})}{T_i}. \tag{1}$$

We assume that the tasks are indexed by criticality—from HI-criticality ones to LO-criticality ones; and HI-criticality tasks are indexed by utilization difference—the larger the utilization difference the lower the index, and utilization difference ties are broken arbitrarily. That is, the HI-criticality tasks are indexed $1, 2, \ldots, n_{\text{HI}}$, and $\delta_i \geq \delta_j$ for any $1 \leq i \leq j \leq n_{\text{HI}}$.

Then, the per mode utilizations of either criticality task set are defined:

$$U_{\text{HI}}^{\text{HI}} = \sum_{i=1}^{n_{\text{HI}}} c_i(\text{HI})/T_i; \tag{2}$$

$$U_{\text{HI}}^{\text{LO}} = \sum_{i=1}^{n_{\text{HI}}} c_i(\text{LO})/T_i; \tag{3}$$

$$U_{\text{LO}}^{\text{LO}} = \sum_{i=n_{\text{HI}}+1}^{n} c_i(\text{LO})/T_i. \tag{4}$$

Mixed-Critical Scheduling. The MC scheduling objective is to determine a run-time scheduling strategy which ensures that: (i) *all* jobs of *all* tasks complete by their deadlines if no job exceeds its LO-WCET; (ii) *all* jobs of tasks designated as being of HI-criticality continue to complete by their deadlines (although the LO-criticality jobs may not) if *any* HI-criticality job requires execution for more than its LO-WCET (but no larger than its HI-WCET) to complete.

Limited HI-Criticality **Behaviors.** As motivated in Sect. 1, in some systems, it could be reasonable to assume that only a limited number N of HI-criticality tasks that may exceed their LO-WCET and reach their HI-WCET simultaneously, where $N \leq n_{\text{HI}}$. In contrast, existing MC analysis usually makes the most pessimistic assumption that all of the n_{HI} HI-criticality tasks may execute beyond their LO-WCET and reach its HI-WCET simultaneously. Even if this could actually happen, it can also be viewed as a special case ($N = n_{\text{HI}}$) under the new MC model we propose in this paper By saying *simultaneously* (or "at the same time"), we mean within any time window of length $\overline{T} = \max_i\{T_i\}$[1]. That is, *at most N* HI-*criticality tasks can require an execution time larger than their* $c_i(\text{LO})$ *within any time window of length* $\overline{T}$. Again, please note that the Vestal model is a special case of our model, by assigning $N = n_{\text{HI}}$.

Determine N. In this paper, we generally assume that the parameter N is a parameter given offline, instead of to be determined online by the scheduler. That is, how to determine N is not the focus of this paper, and we mainly focus on the problem of how to schedule the tasks with a valid schedulability test when N is given as an input parameter. Nonetheless, for the sake of inspiring future work, we also briefly discuss a couple of potential sources for where the N parameter could come from.

First, it could come from physical constraints in the systems. Different set of HI-criticality tasks may be triggered to perform their HI-criticality behaviors by different physical measurements. Such difference may be significant enough so that they cannot have simultaneous impacts on the system.

Second, it could come from contradicting logic control flows in the code. When the code of tasks has branches, which branch is chosen to execute may depend on some global variables. Different task might have the same global variables in their code, and the same global variables control the branch choices in multiple tasks. As a result, it could be logically impossible for some HI-criticality tasks to take their worst branch choices simultaneously. That is, they cannot have their HI-criticality behaviors to have simultaneous impacts on the system.

Third, it could also come from probabilistic analysis if the WCETs of HI-criticality tasks are *independent* [12]. In this approach, the probability of multiple HI-criticality tasks performing HI-criticality behaviors could be calculated as a product of multiple (hopefully small) probabilities for each individual task to perform its HI-criticality behavior. When this product is sufficiently small, the simultaneous HI-criticality behaviors of these tasks could be probabilistically deemed *impossible*.[2] This setting was also considered in [11,13], which more focuses on the various detailed combinations of tasks that may not perform their HI-criticality behaviors. Therefore, a somewhat complicated scheduling approach was studied there. In this paper, we mainly focused on the maximum *number* of such tasks only, and therefore enable the applicability of the relatively simple scheduler, EDF-VD.

3 EDF-VD Schedulability Analysis

In this section, we review a commonly used and adapted MC scheduler, namely EDF-VD [2], which was proposed for the classic Vestal model. We will refine the

[1] When considering a certain time window of length $\overline{T}$, any task τ_i with a partially overlapping scheduling window that experience HI-criticality behavior counts (although it may be already finished by the beginning of the period of interest, or it did not start executing by the end of the period of interest).

[2] Or equivalently, even if it does happen, it is viewed as erroneous, and the system design does not take care of it.

original analysis of EDF-VD to cope with our less pessimistic assumptions, and derive a new schedulability test for EDF-VD under the new model proposed in this paper.

EDF-VD. Similar to the classic EDF scheduler, EDF-VD is a deadline-based, dynamic-priority scheduler. In contrast to EDF, EDF-VD assigns virtual deadlines, which are earlier than the actual deadlines, to HI-criticality jobs. In the runtime, their priorities are determined by their virtual deadlines in the LO-criticality mode; upon a mode switch, their priorities are changed back to their actual deadlines in the HI-criticality mode. Intuitively, the virtual deadlines in the LO-criticality mode provide the room for the HI-criticality tasks to still meet their actual deadlines in the HI-criticality mode, when they occasionally overrun their LO-WCETs.

Let τ denote the MC implicit-deadline sporadic task system that is to be scheduled on a preemptive uniprocessor. Prior to run-time, EDF-VD performs a schedulability test to determine whether τ can be correctly scheduled by it or not. If τ is deemed schedulable, then an additional parameter x is computed for setting virtual deadlines to HI-criticality tasks. Each virtual relative deadline T_i' can be calculated by "shrinking" the actual relative deadline T_i by the scaling factor x.

Next, we describe a schedulability test for EDF-VD under the proposed new model and prove its correctness. Note that, when $N = n_{\text{HI}}$, this schedulability test reduces to the one for the classic Vestal model in [2].

Schedulability Test. First, given an MC task instance, the parameter x is calculated as follows:

$$x \leftarrow \frac{U_{\text{HI}}^{\text{LO}}}{1 - U_{\text{LO}}^{\text{LO}}}. \tag{5}$$

By Theorem 1 (to be presented later), this assignment of x will be able to guarantee the schedulability under LO-criticality mode.

Then, the schedulability under HI-criticality mode can also be guaranteed if the following inequality holds:

$$xU_{\text{LO}}^{\text{LO}} + U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i \leq 1. \tag{6}$$

That is, given an MC task instance, the schedulability test needs to check whether Inequality (6) is satisfied.

The schedulability test returns success if Inequality (6) is satisfied, and failure otherwise.

Upon success, EDF-VD assigns virtual deadline parameters for all HI-criticality tasks as follows:

$$T_i' \leftarrow x \cdot T_i. \tag{7}$$

Correctness Proof. The correctness proof of the above schedulability test contains two parts: (i) all deadlines being met under LO-mode (Theorem 1) and (ii) HI-criticality deadlines under HI-mode (Theorem 2).

Theorem 1. *Under EDF-VD, all tasks meet their deadlines in* LO*-mode (where all jobs complete upon receiving execution time up to their* LO*-WCETs) if*

$$x \geq \frac{U_{\text{HI}}^{\text{LO}}}{1 - U_{\text{LO}}^{\text{LO}}}. \tag{8}$$

Proof: By the density test in [12], $U_{\text{LO}}^{\text{LO}} + U_{\text{HI}}^{\text{LO}}/x \leq 1$ is sufficient to ensure that EDF-VD successfully schedules all LO-criticality behaviors of τ. Theorem follows by rearranging this inequality. □

Lemma 1. *For any period of length t, total demand by* HI*-criticality tasks can not exceed* $(U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i)t$*.*

Proof: It is assumed that HI-criticality tasks are ordered (decreasingly) according to their δ_i values. Consider the scenario that tasks $\tau_1, ..., \tau_N$ requires for executions more than its $c_i(\text{LO})$, than it is obvious that the total demand by HI-criticality tasks can not exceed $(U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i)t$.

We prove by contradiction. Assume there is another scenario with total demand larger than the above mentioned case. We can always identify the difference between this new release pattern with the one we have – by "replacing" one job that is released by one of the tasks from $\tau_1, ..., \tau_N$ by a job released by some task other than $\tau_1, ..., \tau_N$, one at a time. We can not directly add any task since we have reached the maximum number (N) of tasks that can require demands higher than their LO-WCETs. However, since tasks are ordered by their δ_i values decreasingly, the demand of new tasks in the period of interest (between the release and the deadline of the job being replaced) cannot exceed the one created by the original job. Therefore, such "swaps" will always result into a decreasing of the total demand, which contradicts our assumption. □

Theorem 2. *Under EDF-VD, all* HI*-criticality tasks meet their deadlines in* HI*-mode if Inequality (6) holds. In the* HI*-mode, some but no more than N* HI*-criticality job(s) have not completed upon receiving execution time up to their* LO*-WCETs but will complete upon receiving execution time up to their* HI*-WCETs.*

Proof: It is assumed that the reader is familiar with the correctness proof for EDF-VD in [2], so we will skip many parts of the proof that will look identical. We also adopt all notations there: t_f as the first HI-criticality deadline that is missed, 0 as the last idle instant before t_f, $t^* < t_f$ as the mode switch point, η_i denote the amount of execution over the interval $[0, t_f)$ that is needed by jobs generated by task τ_i. a_1 as the release time of the job with the earliest release time amongst all those that execute in $[t^*, t_f)$, and η_i.

The proofs of Facts 1 and 2 remain unchanged due to the minimal set assumption and the same strategy used under LO mode. Regarding Fact 3, here we calculate the maximum total HI-criticality demand over $[0, t_f)$ instead, and then sum the cumulative demand of all the tasks over $[0, t_f)$.

From Lemma 1 we know that during interval $[a_1, t_f)$, the total HI-criticality demand will not exceed $(t_f - a_1)(U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i)$. As a result, we have the following upper bound for cumulative demand of all HI-criticality tasks over $[0, t_f)$:

$$\sum_{\chi_i=\text{HI}} \eta_i \leq \frac{a_1}{x} U_{\text{HI}}^{\text{LO}} + (t_f - a_1)(U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i). \tag{9}$$

From the infeasibility of the instance (due to deadline miss at t_f), we have

$$a_1 + (t_f - a_1)(xU_{\text{LO}}^{\text{LO}} + U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i) > t_f \tag{10}$$

$$\Leftrightarrow \quad (t_f - a_1)(xU_{\text{LO}}^{\text{LO}} + U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i) > t_f - a_1 \tag{11}$$

$$\Leftrightarrow \quad xU_{\text{LO}}^{\text{LO}} + U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i > 1 \tag{12}$$

The contrapositive is exactly Inequality (6), which is sufficient to ensure HI-criticality schedulability by EDF-VD. □

Runtime Behavior. During runtime, if a LO-criticality job of task τ_i arrives at time-instant t_a, then the priority of this job is determined by its deadline $t_a + T_i$, whereas its priority will be determined by its virtual deadline $t_a + T_i'$ if it is a HI-criticality job. If any HI-criticality job executes for a duration exceeding its LO-WCET without signaling completion, the scheduler immediately discards all LO-criticality jobs[3] and executes HI-criticality HI-criticality tasks according to EDF order with their *actual* (instead of virtual) deadlines. Moreover, *idleness* always serves as the trigger to LO-criticality mode of the system.

Additional Discussions. Under the MC scheduling approach, LO-criticality jobs will be dropped in the HI-criticality mode, and any HI-criticality job overrunning its LO-WCET will trigger the mode switch. With the proposed model, this dropping may not be necessary. The following inequality should be examined before the system starts any execution:

$$U_{\text{LO}}^{\text{LO}} + U_{\text{HI}}^{\text{LO}} + \sum_{i=1}^{N} \delta_i \leq 1. \tag{13}$$

If Inequality (13) is true, then actually no mode switch nor virtual deadline is needed. The system can be scheduled by ordinary preemptive EDF scheduler and all deadlines will be met. This result directly follows from Lemma 1. If Inequality (13) is false, we then apply the MC scheduling techniques described earlier in this section, and examine Inequality (6) to verify the schedulability.

[3] An efficient implementation of such a run-time dispatcher may be obtained using the technique described in [2, Sect. V-A], to have runtime that is logarithmic in the number of tasks.

4 Experimental Evaluation

In Sect. 2, we have proposed a new MC system model that specifies the maximum number of tasks N that can simultaneously experience HI-criticality behaviors within *any* time window of length $\max_i\{T_i\}$. With this additional information in the model comparing to the classic Vestal model, we are expecting a "better" schedulability result for EDF-VD under the new model.

In this section, we conduct schedulability experiments to evaluate the effectiveness of the proposed model against the classic Vestal model. Various per-mode utilizations as well as N's are considered in our experiments. The MC task instances in our experiments are generated by the MC task generator described in [3], which has passed artifact evaluation.

In each set of our experiments, the average normalized utilization [4] of the generated task set range from 0.5 to 1 with increasing at step size 0.05. For every average utilization, 1000 task sets are generated and the acceptance ratio indicates how many of them passed the corresponding schedulability test (and thus can be scheduled correctly).

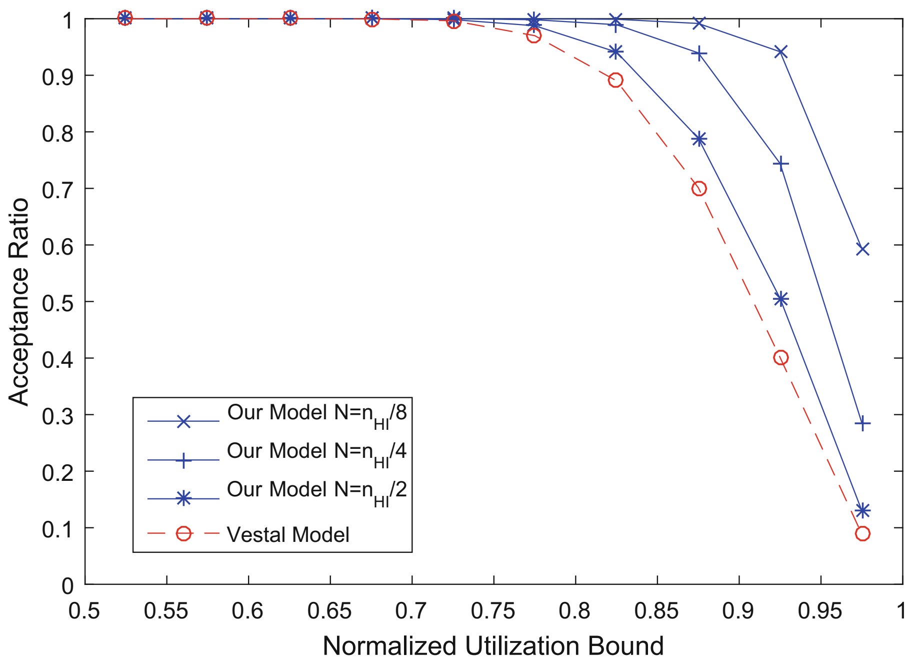


Fig. 2. Schedulability ratio comparison of our proposed model and the classic Vestal model under various N's, with $n_{\text{HI}} = 16$.

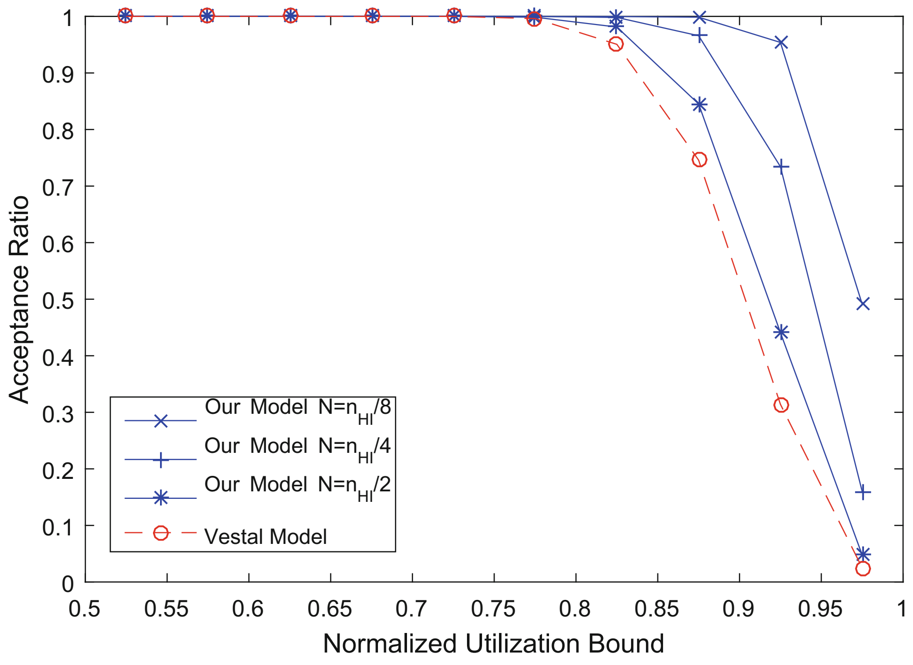


Fig. 3. Schedulability ratio comparison of our proposed model and the classic Vestal model under various N's, with $n_{\text{HI}} = 32$.

Figures 2, 3, and 4 demonstrate the effectiveness of the new model along with the corresponding EDF-VD schedulability test under various settings of numbers of HI-criticality tasks (16, 32, and 64) and sizes of N (i.e., number of HI-criticality tasks that can simultaneously exceed LO-WCETs. It is natral that the acceptance ratios will drop when system is more heavily loaded (with higher utilization). However, we notice that our methods maintains relatively higher acceptance ratio even when normalized utilization gets close to 1.

These results also show that, if less pessimistic assumptions (about the N) can be made, the schedulability can be significantly increased. We do not notice much different in the trends when total number of HI-criticality tasks varies.

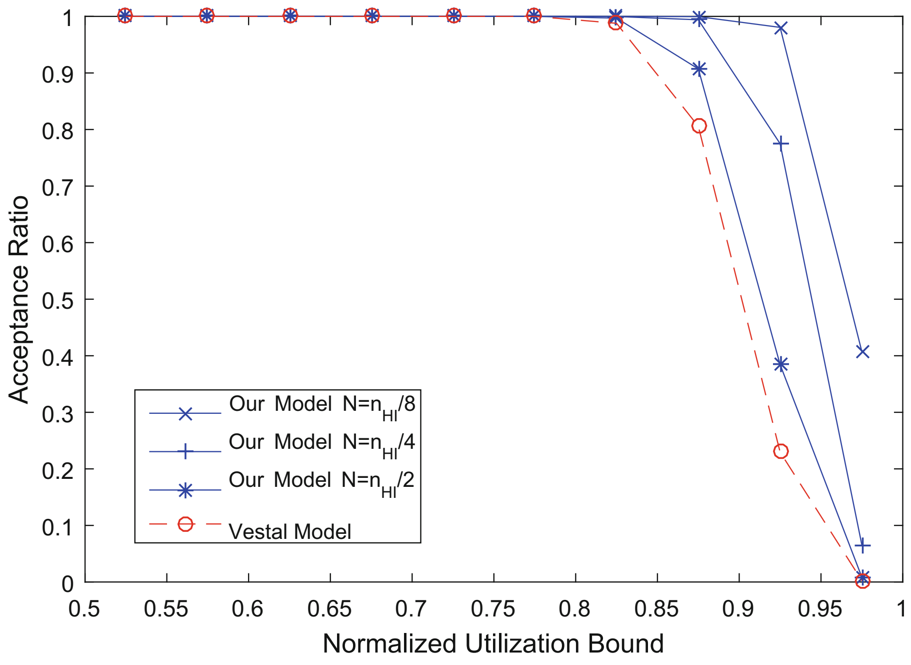


Fig. 4. Schedulability ratio comparison of our proposed model and the classic Vestal model under various N's, with $n_{\text{HI}} = 64$.

5 Conclusion

This paper extends the classic Vestal model for MC scheduling by allowing system designers to specify an additional parameter, representing the maximum number of HI-criticality tasks that may simultaneously exceed their LO-WCETs during runtime. By simultaneously, we mean within any sliding time window of length less than or equal to the maximum period among all tasks. The well-known scheduler, namely EDF-VD, has been studied under the proposed model, and a new schedulability test has been proposed and analyzed. Schedulability experiments have demonstrated that by applying the proposed model in place of the classic Vestal model, significant schedulability improvements can be achieved.

For future work, we would like to consider fixed-priority schedulers under the proposed model, in addition to the deadline-based scheduler, EDF-VD, we considered in this paper. The results may also be extended (at a measurable cost) into multi-processor and/or multi-criticality-level cases.

References

1. Baruah, S.: Mixed criticality schedulability analysis is highly intractable (2008). http://www.cs.unc.edu/~baruah/Submitted/02cxty.pdf
2. Baruah, S., et al.: The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In: The 24th Euromicro Conference on Real-Time Systems (ECRTS 2012) (2012)
3. Baruah, S., Burns, A., Guo, Z.: Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In: The 28th Euromicro Conference on Real-Time Systems (ECRTS 2016) (2016)
4. Baruah, S., Eswaran, A., Guo, Z.: MC-Fluid: simplified and optimally quantified. In Proceedings of the 36th IEEE Real-Time Systems Symposium (RTSS 2015) (2015)
5. Baruah, S., Guo, Z.: Mixed-criticality scheduling upon varying-speed processors. In: The 34th IEEE Real-Time Systems Symposium (RTSS 2013) (2013)
6. Baruah, S., Guo, Z.: Scheduling mixed-criticality implicit-deadline sporadic task systems upon a varying-speed processor. In: Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS 2014) (2014)
7. Burns, A., Davis, R.: Mixed-criticality systems: a review (2016). http://www-users.cs.york.ac.uk/~burns/review.pdf
8. Guo, Z., Baruah, S.: Mixed-criticality scheduling upon varying-speed multiprocessors. Leibniz Trans. Embed. Syst. **1**(2), 3:1–3:19 (2014)
9. Guo, Z., Baruah, S.: Mixed-criticality scheduling upon varying-speed multiprocessors. In: Proceedings of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC 2014) (2014)
10. Guo, Z., Baruah, S.: The concurrent consideration of uncertainty in WCETs and processor speeds in mixed-criticality systems. In: The 23rd International Conference on Real-Time and Network Systems (RTNS 2015) (2015)
11. Hansen, J., Hissam, S., Moreno, G.A.: Statistical-based WCET estimation and validation. In: The 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009) (2009)
12. Cucu-Grosjean, L.: Independence - a misunderstood property of and for (probabilistic) real-time systems. Invited paper to the 60th birthday of A. Burns (2013)
13. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. J. ACM **20**(1), 46–61 (1973)
14. Vestal, S.: Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: The 28th IEEE Real-Time Systems Symposium (RTSS 2007) (2007)

Author Index