Logs and Models in Engineering Complex Embedded Production Software Systems

Logs and Models in Engineering Complex Embedded Production Software Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op woensdag 19 april 2023 om 11:00 uur

door

Nan Yang

geboren te Youxi, China

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

Voorzitter:	prof.dr. O.J. (Onno) Boxma
Promotoren:	prof.dr. J.J. Lukkien
	prof.dr. A. Serebrenik
Copromotor:	dr.ir. P.J.L. Cuijpers
Leden:	prof.dr. M.R.V. Chaudron
	prof.dr. D.Lo (Singapore Management University)
	prof.dr. A. Wąsowski (IT University)
Adviseur:	dr.ir. R.R.H. Schiffelers (ASML)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Logs and Models in Engineering Complex Embedded Production Software Systems

Nan Yang

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and supported by Eindhoven University of Technology and ASML Netherlands B.V., carried out as part of the IMPULS II project. IPA dissertation series 2023-03.

A catalogue record is available from the Eindhoven University of Technology Library ISBN: 978-90-386-5712-7

Cover design: Jiangxue Xu. `www.jiangxue-xu.com`
Printed by ProefschriftMaken.

Acknowledgments

Wonderful! I have finally arrived at the moment to write this acknowledgement letter. Dear everyone, as you can imagine, pursuing a PhD is one of the most challenging endeavors I have undertaken in my life (thus far). Looking back, this has been a long journey, riddled with obstacles, yet simultaneously encompassing incredible prospects and experiences of personal development.

First and foremost, I am immensely grateful to my promotors Prof. Johan Lukkien and Prof. Alexander Serebrenik, my co-promotor Dr. Pieter Cuijpers and my company advisor Dr. Ramon Schiffelers, for their unwavering support and guidance.

Alexander, I still remember the excitement and nerves I felt when I first reached out to you, asking you to be my supervisor for my master's thesis. From that very first email, you were kind, responsive, and willing to help. You guided me through the process of setting up a project at ASML with Ramon and provided me with all the support I needed to complete my master's thesis. You were open, patient, and encouraging, and that was only the beginning of our journey together. Throughout my PhD, you have been a constant source of support and inspiration. There were times when I doubted myself and my abilities, but you never wavered in your belief in me. I often wonder how you manage to support so many people around you while providing prompt feedback and guidance for each of us. I am also inspired by the fact that you are a lifelong learner, always passionate about different languages and cultures! I still remember you told me when I just started my PhD journey: "*The outcome of your PhD is not just your thesis, it is you.*" As I reflect on my experience, I couldn't agree with you more. Alexander, your influence has extended far beyond just critical research, and for that, I am forever grateful.

Johan, thank you so much for giving me the chance to pursue this PhD. I sincerely appreciate all the extra time and effort you have invested in me despite your demanding role as the dean of our department. You are always patient to me when I did not understand the technological terms you used. When I had difficulties in effectively managing my project with multiple stakeholders, you suggest me to seek help instead of trying to solve everything on my own. Your support and encouragement have enabled me to take ownership of my project. Most importantly, I appreciate your constructive feedback, which has consistently pushed me to improve the quality of my research.

Ramon, I cannot thank you enough for leading me to such complex and intriguing research problems at ASML that have truly challenged me and pushed me to grow as a researcher. I still vividly remember our discussions

during our Monday meetings, where you would patiently listen to me explaining what I have learned in the past week. I also recall the times we spent on the bus from ASML to Eindhoven station, where we would continue our discussions. Your boundless enthusiasm and dedication to applied research have profoundly influenced my own research views. Most importantly, I have learned so much from you about how to connect dots and pieces together. Your ability to see the bigger picture and synthesize complex ideas is truly remarkable. Thank you for being such a critical and supportive mentor!

Pieter, thank you for teaching me how to present my work! I still remember our weekly meetings where we had countless discussions about research methods. At that time, I was overwhelmed and struggling to understand the different research philosophies, and to navigate through all these differences. Looking back, it is clear to me that these conversations have played a significant role in shaping my research stance and understanding researchers from different fields. Thank you for your willingness to share your insights and perspectives with me.

Further, I would like to express my great thanks to the members of my doctoral committee, Prof. David Lo, Prof. Michel Chaudron, and Prof. Andrzej Wąsowski for their valuable feedback and advice, which have helped to refine and strengthen my research.

I would like to acknowledge the 43 software developers who participated in my interviews. Although I cannot mention their names for confidentiality reasons, I deeply appreciate their contributions to my research.

I am grateful to have had the opportunity to work with the amazing colleagues and friends from the IRIS and SET research groups during this joint PhD project. I want to express my sincere thanks to the group leads, Prof. Mark van den Brand, Prof. Michel Chaudron, and Prof. Nirvana Meratnia, for providing me with an inspiring and supportive research environment. I also want to extend my gratitude to my wonderful office mates, Jinyue Cao, Leila Fatmasari Rahman, Luis Perez Rey, Iram Bibi, Ali Mahmoudi, Tianyu Bi, and Bram van Berlo. Our coffee breaks were always a highlight of the day!

Many thanks to my lunch mates, Priyanka Karkhanis, Sangeeth Kochanthara, Ana Maria Sutîi, Kousar Aslam, Maurice Laveaux, Mahdi Nikoo, Nathan Cassee, Lina Ochoa Venegas, Hossain Muctadir, David Manrique Negrin, Rick Erkens, Satrio Rukmono, Lars van den Haak, Gizem Karagoz, Mazyar Seraj, Felipe Ebert, Mauricio Verano Merino, Srinidhi Srinivasan, Daniela Girardi, Olav Bunte, Weslley Silva Torres, Josh Mengerink, Yaping Luo, Jan Martens, Mark Bouwman, Vasilis Tsouvalas, Shengyuan Yan, and Saeed Khalilian Gourtani. It was always a pleasure to share lunch with such a diverse and supportive group of people.

Moreover, I am grateful for the friends who supported me throughout the writing process. David, thank you for our coffee and bubble tea chats that provided a much-needed break from the thesis. Lina, Sangeeth, and Priyanka, I feel lucky to have had you by my side throughout the writing journey. Your constant encouragement and support helped me push through the tough times. Satrio, thank you for your insightful advice for my thesis. Nathan, I want to thank you for organizing all the wonderful PhD events that brought us all

together. I want to extend my gratitude to Gizem, Nidhi, and Yimi for being a constant source of girl power!

Since my master's graduation project, I had the great opportunity to conduct research at ASML SW Research. I am immensely grateful to my colleagues there for providing constructive feedback on my work. Special thanks to Kousar Aslam, Thomas Neele and Ruben Jonk for all the interesting discussions, presentations and gatherings! I would like to express my sincere gratitude to Dennis Hendrik for his attendance at my weekly meetings, his invaluable feedback, and his diligent paper reviews. Dennis, thank you for being such a supportive presence in my academic journey since my time as a master's student! I would also like to extend my gratitude to Berk Aksakal for the pleasure of supervising your graduation project. Thank you also for your help with the SLR study!

In addition to my research at ASML, I have been fortunate to collaborate with Prof. Bram Adams from Queen's University and Isabella Ferreira from Polytechnique Montréal. This opportunity has been eye-opening, and I've gained valuable insights and perspectives through our discussions. Thank you, Bram and Isabella! Also, I would like to thank you again, Alexander, for arranging this collaboration and for going above and beyond to help me broaden my horizons.

Doing a PhD during a pandemic adds extra challenges. When I started my PhD, attending academic conferences in person was still possible. I cherish the memories of those travels, which allowed me to meet some amazing people. Davide, you made my first conference in Hangzhou so much less intimidating, and I am so glad that we later met again in Montreal for ICSE-2019. It was a pity that we cannot meet in Korea for ICSE-2020 due to Covid, but I am sure we will meet somewhere soon! Daniela, thank you for being so kind when we met in Lugano for the SIESTA summer school, and I had a wonderful time when you visited us in Eindhoven.

Learning Dutch was one of the things that helped me get through my PhD during the pandemic, and I am incredibly grateful to the people who dedicated their time and effort to help me achieve this. Therefore, I want to give a special shoutout to Sander de Putter, Rick Erkens, Jolande Matthijsse, Geert van Kollenburg, Marijn van Knippenberg and Richard Verhoeven. Thank you for welcoming me into your world and helping me prepare for my Dutch exams! Your kindness made me feel at home, even though I am thousands of miles away from China. I also want to express my appreciation to my friends whom I met through language exchanges. Kim, I have enjoyed our weekly Dutch-Chinese hour and I always admire your passion for learning. Cécile, I feel so lucky to meet you in Leiden and I will always remember the enjoyable conversations and the cultural insights we exchanged. Dankjewel allemaal!

None of this could have been possible without my dear friends. Lu, Linhao and Xiaolei, thank you for being a source of positivity in this journey! Many thanks to my friends Sinan, Qinxin, and Hyunseon in Leiden for their kindness and warmth during my stay, which helped me revise this thesis. Furthermore, I would like to use this opportunity to express my gratitude to my dear gang: Huiyi, Shidong, Yuyang, Bin, Tianyu, and Jiangxue. You have been my anchor here in the Netherlands, and no matter where I wander, I always know I can find

my way back to you. Huiyi, thank you for taking care of everyone around you with your warm heart and amazing food. Shidong, thank you for being so calm and offering hugs when I was in difficult situations. I was lucky enough to see you two finish your PhD and I have learned a lot from watching your journey. Yuyang, thank you so much for always having my back and being supportive for my crazy ideas. I honestly don't know how I would have survived without your kind assistance over so many years. Bin, words cannot fully convey how grateful I am for your support and encouragement during that difficult period when I was on the verge of giving up. Having you by my side was a source of strength and motivation. Tianyu, I've had so much fun poking fun at you, and I'm grateful that you never take it too seriously. Jiangxue, special thanks go to you for designing the amazing cover for my thesis!

One of the most incredible things that has happened to me is the friendship I have with you, Chang. I consider myself fortunate to have had you as my roommate when we studied in Shanghai, and even more fortunate to still be each other's best friend after 12 years! Thank you for inviting me to be your paranymph for your defense, and for all the wonderful trips and memories we have made over the years.

My endless thanks go to all my other friends!

Finally, I want to pour out my heart to my beloved family for being the backbone of my life. 铭姐，谢谢你所有的远程陪伴！妈妈，谢谢你尽你所能给我最好的教育和支持！

Nan Yang

Contents

List of Figures

List of Tables

1

Introduction

Embedded systems are an essential part of production and manufacturing, providing a strict control and automation of the processes to ensure the quality of final products. One can find embedded systems in many production industries, such as agriculture and semiconductor industry. The trends in modern production are characterized by mass customization, high variability of machine configurations, and extensive technical updates to meet the ever-changing requirements and specifications of production [393]. Such systems are comprised of mechanical, electronic, software and other components, structured in different abstraction layers, and implemented with multiple technologies. As discussed in literature [138, 212, 393], embedded production systems impose specific challenges in the process of software development and maintenance. The most frequently discussed challenges in the literature include, but are not limited to:

- *explicit and implicit dependencies between thousands of software and hardware components that are operating with each other.* It is challenging to assess the impact of a change to one component on other components of a system.
- *critical performance requirements of systems.* The systems that drive production often have critical performance requirements that demand real-time behavior. Even minor software changes can significantly affect the timing of these systems, impacting production throughput.
- *concepts and designs related to different disciplines.* The software for embedded production systems often encompasses concepts and designs from various disciplines such as physics and chemistry, which makes comprehending system behavior challenging. As time passes, developers with deep domain knowledge may have left the project or the company, further complicating the maintenance of legacy code.

Hence, software development and maintenance of embedded production systems requires tailored software engineering methodologies, processes and

tools due to the real-time, heterogeneous and multidisciplinary nature [138, 393]. The general software engineering methodologies may not have the features to address the challenges raised by these requirements. For example, empirical studies have shown that embedded systems require different tools in testing [128, 360] and continuous integration [297].

To propose new and improve existing methodologies, processes, and tools suitable for embedded production systems, we believe that it is essential to obtain practical knowledge from a real-world context where software developers are involved. Therefore, we aim to increase our understanding of software engineering practice for embedded production systems by empirically studying the question:

> **RQ**: How do software developers engineer the software of embedded production systems?

In this thesis, we study this question by collaborating with ASML, a leading manufacturer of lithography machines for the semiconductor industry. The industrial collaboration allows us to adopt the case study methodology—a widely used empirical method aimed at investigating phenomena in their context [324]. Machines developed by ASML are typical examples of embedded production systems. The software system of these machines consists of a large code base of more than 40 million lines of code[1] that implements the concepts and requirements related to different disciplines (e.g., physics, mechanics and electronics). The system has multiple layers, consisting of thousands of components that are developed by groups of engineers from different engineering backgrounds. These components interact with each other, and play various roles in the architecture of the system, ranging from high-level production controllers to hardware drivers.

1.1 Research Questions

We study a number of topics that were all considered of immediate interest to our main stakeholder ASML. Figure 1.1 shows the research studies presented in this thesis. Note that the chapters are not listed in a chronological order, but organized in such a way to highlight the links between the topics studied in this thesis, providing readers a way to read and interpret our studies. As can be seen from the figure, we studied several topics centering around logs and models in the engineering process for embedded production systems. Logs and models are of great interest to ASML because they are essential software engineering artifacts used in ASML [332].

As stated in Lehman's laws of software evolution [218], real-world systems (i.e., E-type systems) are undergoing changes due to various maintenance activities such as bug fixing, new feature implementation and refactoring. Diving into the large-scale and ever-changing code base, and reasoning

[1] `https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/company/events/conferences/matlab-conference-benelux/2015/proceedings/facing-moores-law-with-model-driven-r-and-d.pdf`. Accessed: March 2022

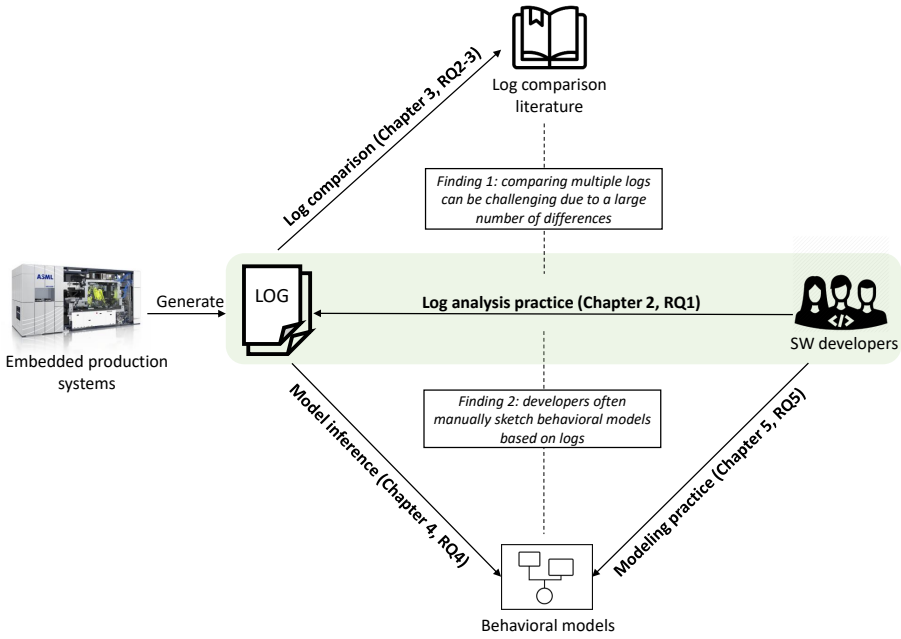


Figure 1.1: Research overview

about the execution paths of programs can be difficult. Logs that record runtime execution of software systems become an essential source of necessary information during various activities such as bug fixing [421], anomaly detection [370], and system monitoring [433]. **Similarly to many other complex systems, systems at ASML generate a large amount of log information that capture the runtime behavior of the systems. ASML has a great interest in leveraging the information contained in logs for software engineering activities with automated tools.**

This is why our first research question, RQ1 introduced in Section 1.1.1 and discussed in Chapter 2, aims at understanding the current industrial log analysis practices. Based on the observations that developers need advanced techniques for comparing multiple logs, our RQ2 and RQ3 (introduced in Section 1.1.2 and discussed in Chapter 3) investigate existing log comparison techniques in the literature.

As the observations from the study of RQ1-3, models are used in developers' practice and log comparison techniques to abstract log information. **This is inline with ASML's transition from traditional software engineering to model-driven software engineering (MDSE).** In fact, the adoption of MDSE is observed not only in ASML but also widely in the embedded industry [12, 228, 229]. In MDSE, models are the primary artifacts of the development and maintenance process. Abstracting software behavior with behavioral models can further enable formal analysis and verification, as well as code generation. That is, developers can specify the behavior of software with models, verify the correctness of software behavior using model checking techniques, and then

generate code from the verified models. The subsequent maintenance changes such as bug fixing and feature implementation are all made to models. **The increasing adoption of MDSE and ongoing transition calls for research efforts to study the practices, processes and tools of MDSE.** That is why, our RQ4, introduced in Section 1.1.3 and discussed in Chapter 4, studies how to infer models from existing code-based systems, aiming at helping companies transit from code-based engineering to MDSE. Although model inference techniques have been improved by researchers over the years, industry still relies on developers to create models manually. We therefore study RQ5 (introduced in Section 1.1.4 and discussed in Chapter 5), aiming at understanding how developers model software in practice.

1.1.1 Log Analysis Practice: An Exploratory Study

Due to the execution information they provide, execution logs are considered to be essential inputs for software analytics tools and processes that aim at helping developers deal with the complexity of large-scale systems [439]. Log analysis techniques have been proposed to aid developers in such software engineering tasks as program comprehension [325], test generation [308], and change comprehension [19, 43, 254].

Understanding how developers obtain required information when maintaining complex embedded software is essential to propose useful tools; if the required information and actionable insights could be easily extracted and provided by tools, developers could focus their effort and time on the maintenance tasks, rather than on searching for relevant information [439]. Therefore, we state:

> **RQ1**: How do developers use logs in engineering embedded production software?

In order to answer this question, we conduct an exploratory study, which consists of two parts. First, we interview 25 developers from ASML [427] about the types of logs that they use, purposes for which they use logs, types of log information that they need, challenges that they face, and tools they use. To understand to what extent our findings at ASML are transferable to other companies, we further interview 14 developers from four other companies. The discussions from this exploratory study provide insights in the challenges developers face and the information that they need, as well as possible tool support for log analysis.

Among all the observations obtained from this exploratory study, in this thesis we further zoom in on two based on ASML's interest. First, we observe that developers often compare multiple logs for various maintenance tasks with text-based comparison tools. However, comparing logs is challenging due to the lack of tools that can deal with the complex nature of software (e.g., interleaving introduced by concurrent executions). To provide more insights into this problem, in this thesis we explore existing log comparison techniques (Section 1.1.2). Another interesting observation from this exploratory study is that developers often manually sketch behavioral models (e.g., state

machines and sequence diagrams) based on log information to obtain an abstract representation of executed behavior. This observation, in hindsight, supports the idea that developers at ASML might benefit from the use of MDSE for their software development and maintenance practices. However, the transition from traditional software engineering to MDSE raises a series of challenges. In this thesis, we zoom in on two challenges about inferring models from an existing code base (Section 1.1.3) and modeling software with MDSE tools (Section 1.1.4).

1.1.2 Log Comparison: Understanding the State of the Art

As we observe in the exploratory study (RQ1), developers use text-based tools for comparing multiple logs and face the challenges in identifying actionable insights from a large number of differences between logs. This is consistent with the observations at Google [146] and Microsoft [44]. However, over the years, researchers have proposed many log comparison techniques to address different challenges for different maintenance activities. As the first step required to identify the gap between the state-of-the-art and practices, it is important to understand what techniques exist and what limitations these techniques have.

Therefore, we study:

> **RQ2**: What are the existing log comparison techniques?

As indicated by developers from ASML and Google [146], **a major challenge that developers experience is that the trivial and unimportant log differences overshadow the important information that points to the problem.** For example, when using text-based tools to compare logs for identifying the root cause of bugs, one of the challenges is to deal with the interleaving of events caused by concurrency. It is interesting to study how the state-of-the-art log comparison techniques address the challenges faced by developers. Therefore, we further dive into the log comparison techniques that address the industrial challenges. We ask:

> **RQ3**: How do existing log comparison techniques address industrial challenges?

1.1.3 Model Inference: Combining Active and Passive Learning

To gain the potential benefit that MDSE promises, practitioners are still facing many challenges [228, 274, 386, 409]. One of the challenges is the transition from traditional software engineering to MDSE because the existing code base is typically huge [346]. Manually creating models is very time-consuming, as it requires developers to comprehensively understand the behavior of the complex systems. **A more effective, but yet challenging method would be automatically inferring models from software and its artifacts.**

Model inference techniques can be broadly classified into those based on static analysis and dynamic analysis. Static analysis is an offline approach

which does not require the execution of software systems. Instead, it examines the code and all the execution paths to construct models [84]. Typically, static analysis is time-consuming and suffers from scalability issues when dealing with large-scale software systems [176]. The learned model is often an over-approximation of software behavior. Dynamic analysis, in contrast, constructs a behavioral model based on observations by executing the systems. It is usually more efficient as it learns an under-approximation of software behavior (i.e., incomplete models). Moreover, dynamic analysis can capture how software systems interact with their environment, which is not identifiable in static analysis. In this thesis, we investigate dynamic model inference techniques because embedded production systems are usually not standalone but integrated to work with their environment.

There are two types of dynamic model inference techniques in the literature, namely active learning and passive learning. In 1987, Angluin proposed the L* algorithm [23] which lays the foundation for active learning techniques. Active learning techniques dynamically construct models based on iterative interactions with systems, and refine the hypothesized models with exhaustive testing or random testing. The completeness of the learned model is guaranteed under the assumption that the counterexamples differentiating the hypothesized model from the system can be found via testing. However, the required number of tests to be executed grows exponentially with the size of the system. The execution of such large set of tests can be very time and resource consuming [383].

Passive learning, on the other hand, infers models from logs that capture software behavior at runtime [134, 394]. The learning process is more efficient because interactions with systems are not required. However, since logs are often collected from the executions of limited use cases, the learned models often capture only partial behavior of systems.

Combining the insights provided by these two groups of studies, we learn that active and passive learning techniques have their own limitations and strengths. It is yet an interesting research problem to explore whether these two types of techniques can be combined, exploiting the benefits of one to aid the other. Therefore, we state:

> **RQ4**: How to combine active and passive learning techniques to infer models from code and logs?

1.1.4 Modeling Practice: Why Developers Violate Guidelines

Another striking challenge in MDSE is the lack of suitable modeling tools to facilitate the use of models in such a large-scale and heterogeneous code base [332]. As the result of the ongoing transition to MDSE, the share of the models is increasing in the code base of complex embedded systems. Even though MDSE promises a lot of advantages, it requires suitable modeling tools. However, as reported in literature [407], the majority of modeling tools fail to support development activities by forcing developers to work in a way that fit the tool instead of making the tool fit the people. **This observation**

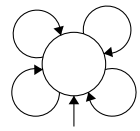


Figure 1.2: A single state machine (also called a flower model). The circle represents the single state, and the arrows going from and to the same state represent the transitions. The incoming arrow indicates the initial transition into this state.

emphasizes the importance of understanding how developers use models and what challenges they face in practice.

There are several existing observation studies on modeling practice [205, 306]. These studies observe how students model software using Unified Modeling Language (UML). As reported in the SE literature, developers often have valid reasons to violate SE guidelines (e.g., coding guidelines), and the violations reveal the limitations of software engineering tools and the challenges faced by developers [69, 291, 382]. **Inspired by these studies, we are interested in identifying the challenges of MDSE practices through the lens of how developers adhere to modeling guidelines and why developers violate guidelines if they do so.**

In this study, we start the exploration of this question with a wide-spread modeling guideline: a state machine model is only meaningful if it contains more than one state, and if each state represents different behavior. The intuition behind this guideline is that a model should contain non-trivial information, otherwise it merely clutters the presentation of ideas [20]. Single-state state machines (SSSMs)—affectionately known as "flowers" due to their graphical representation shown in Figure 1.2—violate this recommendation. This type of model has only one state with self transitions, thus it does not describe the change of software states. We observe at ASML that developers use SSSMs even though the modeling guideline suggests not to do so. The violation triggers the question:

RQ5: Why do developers use single-state state machines in practice?

1.2 Thesis Outline and Origins of Chapters

This thesis contributes to the body of research on understanding developers' practice for dealing with large-scale embedded software systems by obtaining empirical insight from software and software processes in companies, and understanding current state of techniques. Figure 1.1 shows the relation between the topics that we investigated, and the chapters that they correspond to.

Chapter 2: Log analysis practice. In this chapter, we answer RQ1. We report an interview study with 25 developers from ASML and a replication

study with 14 developers from other four companies. We present the identified challenges and information needs, as well as possible tool support for log analysis. This chapter is based on:

> [425] Yang, N., Cuijpers, P.J.L., Hendriks, D., Schiffelers, R.R.H., Lukkien, J., & Serebrenik, A. An interview study about the use of logs in embedded software engineering. 2023. Empirical Software Engineering, 28. Special issue, selection of papers from ICSE-SEIP 2021.

a special issue extension of

> [427] Yang, N., Cuijpers, P.J.L., Schiffelers, R.R.H., Lukkien, J., & Serebrenik, A. An interview study of how developers use execution logs in embedded software engineering. In IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (pp. 61-70). 2021. IEEE.

Chapter 3: A literature study of log comparison techniques. This chapter addresses RQ2 and RQ3. We provide an in-depth analysis of 81 papers about log comparison techniques used in software engineering. We study the SE activities these techniques aim to help, the methods of comparison and the evaluation of these techniques, and the methods provided to address industrial challenges presented in Chapter 2. This chapter is based on

> [429] Yang, N., Hendriks, D., Lukkien, J., & Serebrenik, A. A Literature Review of Log Comparison Techniques for Software Engineering. ACM Transactions on Software Engineering and Methodology. Manuscript under review.

Chapter 4: Combining Active and Passive Learning for Model Inference. In this chapter, we answer RQ4. First, we study 218 software components to provide empirical evidence on the trade-off between learning time and completeness achieved (i.e., the more time is spent, the more behaviors can be learned via interactions with systems) in active learning. The required number of interactions for learning an accurate model grows exponentially with the number of states of the system under study. To solve the problem, we introduce a hybrid solution that enhances active learning techniques with logs and models learned from logs by passive learning techniques. We evaluate the proposed hybrid solution with industrial components from ASML. This chapter is based on the following publication:

> [424] Yang, N., Aslam, K., Schiffelers, R.R.H., Lensink, L., Hendriks, D., Cleophas, L., & Serebrenik, A. Improving model inference in industry by combining active and passive learning. In IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (pp. 253-263). 2019. IEEE.

Chapter 5: Use of Single-state State Machines in Practice. In this chapter, we address RQ5. First, we present the study of mining 1500 state

machines from ASML. By interviewing developers, we then identify the reasons for which developers use singe-state state machines in practice. We then study when these single-state state machines were introduced in the systems. Lastly, we provide implications based on our findings for researchers, practitioners and tool builders. This chapter is based on:

> [428] Yang, N., Cuijpers, P.J.L., Schiffelers, R.R.H., Lukkien, J., & Serebrenik, A. Single-state state machines in model-driven software engineering: an exploratory study. 2021. Empirical Software Engineering, 26(6), 1-46.

a special issue extension of

> [426] Yang, N., Cuijpers, P.J.L., Schiffelers, R.R.H., Lukkien, J., & Serebrenik, A. Painting Flowers: Reasons for Using Single-State State Machines in Model-Driven Engineering. In Proceedings of the 17th International Conference on Mining Software Repositories (pp. 362-373). 2020.

Chapter 6: Conclusions. In this chapter, we revisit our research questions and propose future research directions.

Suggested Methods of Reading

Figure 1.1 shows the overview of the chapters presented in this thesis. These chapters are self-contained and can be read independently. In each chapter, we present the introduction, research questions, methodology, result and discussion. We do not provide a separate chapter on related work. Instead, we discuss the related work in each chapter.

2

Log Analysis Practices for Embedded Production Software

Execution logs capture the run-time behavior of software systems. To assist developers in their maintenance tasks, many studies have proposed tools to analyze execution information from logs. However, it is as yet unknown how industry developers use logs in engineering embedded production systems. Without understanding how developers use logs, the proposed log analysis tools might not be able to meet developers' expectations and needs. In this chapter, we present our exploration about log practice in industry (the RQ1 highlighted in Figure 2.1). We first present the study conducted in ASML, followed by a replication study conducted in four other companies.

2.1 Introduction

Execution logs, produced by software systems at runtime, capture the dynamic aspects of the software. Log analysis tools have been proposed to aid developers in such software engineering tasks as program comprehension [325], test generation [88, 308], and change comprehension [19, 43, 254]. However, researchers have provided empirical evidence that log analysis tools are not necessarily effective and applicable when dealing with real-world problems [258]. Legunsen et al. [217] studied the effectiveness of specifications mined from execution traces in the context of bug-finding. The authors manually analyzed runtime violations of the specifications for 200 open-source projects and found that most of the violations to the specifications are false alarms (i.e., not real bugs). Another problem reported by Mashhadi et al. is that state-of-the-

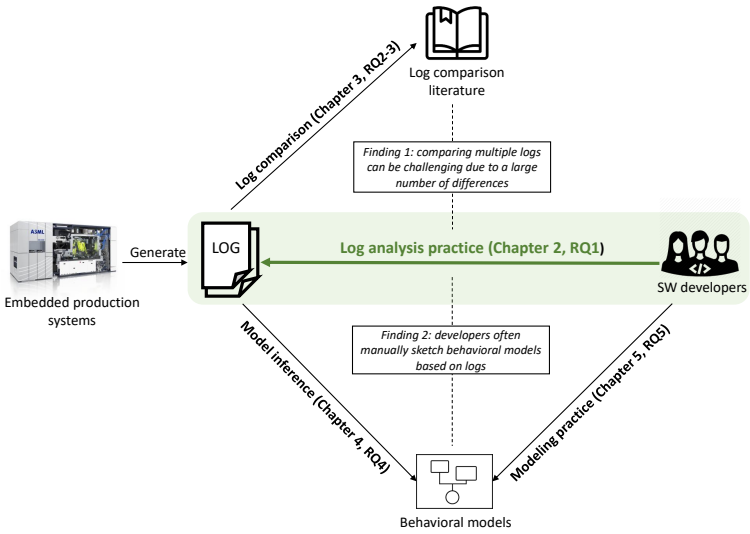


Figure 2.1: Research overview (RQ1)

art log analysis tools failed in processing the large volume of logs produced by a large-scale embedded system [258].

We believe that understanding how engineers analyze logs and what information they need is essential to design better log analysis tools and logging process. Li et al. [224] studied the benefits and costs of logging from developers' perspectives in the context of open-source software development, suggesting better automated logging tools. Barik et al. [44] identified the tensions that emerge in data-driven cultures as event logs are used by a variety of roles including non-engineering roles (e.g., a program manager) at Microsoft, calling for tools that assist non-technical team members in analyzing logs. However, there is no empirical study on developers' log analysis practices in embedded software engineering. Often, embedded software engineering needs specifically targeted tools [138]. Embedded systems are often implemented as concurrent systems, have real-time constraints and are mapped directly on real hardware. These features of embedded systems have raised challenges in software testing [360], modeling [344], and architecture design [24], which opens up questions of how these features influence log analysis practices, what kind of challenges are raised in practices and how developers deal with these raised challenges. Therefore, we focus on how developers analyze logs in embedded software engineering, with the aim of identifying developers' needs for future research on the techniques that are applicable to aid developers in performing their maintenance tasks.

In order to understand how we can improve analysis tools for embedded software engineering, we need to understand **what information developers need** from execution logs (RQ1.3) and **what tool support could be useful** (RQ1.4). We believe that if the required information could be easily provided by tools, developers could focus their effort and time on the maintenance

tasks, rather than on searching for the information. However, the expectations developers have about tools also depend on the context of use. Therefore, first, we need to understand **the types of logs** developers use (RQ1.1) and **the purposes for which developers analyze logs** (RQ1.2).

To answer our research questions, we first conducted an exploratory case study at ASML, a company that develops lithography systems for the semiconductor industry by interviewing 25 software developers. Table 2.1 summarizes the findings obtained in this exploratory case study.

Table 2.1: Findings by Research Question 1.1-4

Research Question	Main findings
RQ1.1	Developers use four types of execution logs: *high-level machine actions and errors*, *low-level execution details*, *performance data*, and *business-critical data*.
RQ1.2	Logs are primarily used for *analyzing software issues*, and also used for *test code development* and *requirement reverse-engineering*.
RQ1.3	13 types of information are searched for in the execution logs, with the most frequent types being: *propagation of errors across systems*, *timestamps associated with log lines*, *data flow*, *interaction of software components*, and *differences between multiple executions*.
RQ1.4	Challenges in log analysis include: *lack of domain knowledge*, *lack of familiarity with code base and software design*, and *presence of concurrency*. Developers expect tools to help handle complexity by adding *multi-level abstractions* to logs and *comparing multiple logs on different levels of abstraction*.

Our interview at ASML is a case study. As any other case studies, it inherently suffers from threats to external validity. Hence, to increase the external validity, we then extend this study by replicating it at four other companies. With this replication study, we aim at understanding **to what extent our findings at ASML can be confirmed** at other companies (RQ1.5). We would like to not only confirm our previous findings, but also explore the scope of the results. To achieve the confirmatory and exploratory goals, we conducted 14 interviews with engineers from three embedded software companies and one company which develops general applications. By involving these two types of companies, we attempted to identify aspects that are specific to embedded software companies and juxtapose them with findings for companies from other domains. The results show that **the practices at ASML are not company-specific (RQ1.5)**. We found that most findings obtained at ASML (e.g., the challenges raised by concurrency) largely resonate with engineers at other embedded software companies, while some findings are shared by all the companies (e.g., challenges related to log quality) including the company which develops general applications.

We have also collected new insights from this replication study. To address the challenges in log analysis, most of the interviewees shared that it is important to resolve several trade-offs in logging. For example, formalized and automatic logging on one hand can help companies govern log quality, and subsequently facilitate the analysis of logs and the evolution of logging code, but on the other hand, it reduces the freedom of developers in logging what they need and want. Moreover, with this replication study, we collected empirical evidence that the evolution of logging code has raised challenges for maintaining the artifacts that depend on the generated logs (e.g., analytical tools or a knowledge database based on logs).

Based on our results from the study at ASML and its replication, we synthesize the main scenario of software logging, discuss the contextual factors (e.g., programming languages), and formulate implications for practitioners, researchers and tool builders about log analysis and logging practice. For example, we suggest researchers to **study the co-evolution of logs and log-dependent entities to ease any software engineering activities and techniques** that depend on the generated logs (e.g., log-based testing, pattern recognition and matching, and log analysis and differencing). For tool builders, we suggest **developing tools that can help developers comprehend systems in an abstract way, categorize log differences for providing actionable insights and link different types of logs to provide a complete picture of executions**. For practitioners, we suggest **a series of logging guidelines**, such as defining logged information with the stakeholders of logs.

The remainder of this chapter is organized as follows. In Section 2.2, we present our exploratory study at ASML. Next, in Section 2.3, we report on our replication study at four companies. Based on these two studies, we synthesize the findings in Section 2.4. We then discuss the implication of our work in Section 2.5. We discuss threats to validity in Section 2.6. Finally, we conclude in Section 2.7.

2.2 Use of Logs at ASML

In this section, we present our exploratory study at ASML about log analysis. We start with our methodology (Section 2.2.1). The results of this study that answer our research questions are then presented in Sections 2.2.2 (RQ1.1), 2.2.3 (RQ1.2), 2.2.4 (RQ1.3) and 2.2.5 (RQ1.4).

2.2.1 Methodology

To understand how software developers use logs in the embedded systems industry, we conducted a case study [324]. As our research questions differ from previous work [44, 224], we opted for an exploratory rather than a confirmatory study.

2.2.1.1 Context

Our study is part of an ongoing collaboration with ASML who develops high-tech production systems for the semiconductor industry. The division that we work with is responsible for components implementing the supervisory control and metrology of the manufacturing process. Control and metrology have

become the backbone of many high-tech systems (e.g., optical measurement systems and autonomous vehicles) due to the growing complexity and the demanding precision [197].

The software components developed by this division form a paradigmatic subsystem [117] that coordinates machine actions and measurements, as well as calibration of the systems based on the performed measurements. The subsystem consists of multiple processes collaborating with each other via inter-process communication.

The division provides a typical context of embedded software engineering; the (sub)system is implemented not only by software engineers but also engineers from different disciplines (e.g., mechanical or electrical engineering). Similar to other complex embedded systems [29], the execution of such software systems requires the physical layers to be present or simulated. The in-house execution of such software systems requires either a simulator called DevBench, or an environment called TestBench in which physical layers are present.

The system is implemented with several languages. The interviewees use general purpose programming languages C/C++ and Python. In addition, the division has been adopting model-driven engineering (MDE) to design the components that are responsible for controlling machine actions and production processes. Developers design these components using a state-machine-based modeling language called ASD [62]. The correctness of software components is verified using a built-in model checker. The source code of these components is automatically generated from these state machine models.

2.2.1.2 Semi-structured Interviews

We opt for semi-structured interviews as they allow us to discuss prepared questions and ask follow-up questions exploring interesting topics that emerge during interviews [53]. We start the interview with a brief introduction of our research and provide our definition of execution logs: *textual files that record dynamic information produced by the execution of software systems*.

Table 2.2 shows our interview guide. In adherence to best interviewing practices [53], we conducted a pilot interview with a developer from the same division to examine our interview settings and questions. The pilot interview took one hour and led to the rephrasing of several questions. The study design was approved by the ethical review board of the Eindhoven University of Technology and ASML.

2.2.1.3 Interview Participants

The selected division has seven software development groups. Each group is responsible for the development of multiple components. We contacted the group leads from these seven groups to recruit software developers. We encouraged the group leads to take into account the diversity of developers' education background, development role and gender. Our invitation was accepted by 25 software developers (see Table 2.3). In the beginning of the interviews, to establish mutual trust, we stressed that the interviewees' identity will not be disclosed, and their answers will not be shared with their supervisors.

Table 2.2: Interview guide

Background
1. What is your job title?
2. What kind of systems is your group responsible for? What is your role and responsibility within the group?
Type of logs (RQ1.1)
3. Do you use any execution logs that capture the run-time behavior of software?
4. How are these logs commonly called in your team?
5. How do you obtain these execution logs?
Purpose of log analysis (RQ1.2)
6. For what purposes do you use them?
7. How often do you analyze logs for your purposes?
Information needs (RQ1.3)
8. What information is in log X (i.e., the log the developer has mentioned)?
9. What information in log X helps you for your work?
10. How does the information in log X help you?
11. Can you describe the procedure of a task in which log X is used?
Tool support (RQ1.4)
12. What tools do you use for analyzing execution logs?
13. How do you use these tools?
14. What are the most challenging steps in your log analysis practices?
15. How do you cope with these challenges?
16. What kind of tools would you like to have for helping you analyze logs?
17. How would you like to use these tools?
Ending
18. Having discussed some topics about log analysis, would you like to add some thoughts?
19. What is your year's of experience as a software developer?
20. What is your education background?

2.2.1.4 Data Collection and Analysis

We collected data by recording the audio and making the transcripts. We coded the transcripts [53] using the *ATLAS.ti* data analysis software. Our coding process consists of three steps. First, we performed open coding. We constantly compared and refined codes that emerge from this process. Similar codes were then grouped into categories. Second, we conducted axial coding to make connections between codes or categories. Finally, these codes and categories were grouped into the topics derived from our research questions. According to Strauss and Corbin [361], theoretical saturation is reached when no new insights emerge. Hence, instead of having a strict sequential order of data collection and analysis, we interleaved these steps. The codes and categories emerged as the data is analyzed and helped us to examine whether theoretical saturation was reached. We consider that the saturation is reached when no new codes are found. With these 25 participants, we reached the saturation as we did not observe new codes in the last four interviews. We present our explanation of the derived codes in the following sections. The codes are explained with quotes of developers. We give an ID for each quote to help readers link these codes and the explanations. An ID has a format of PX-Y where PX indicates the participant ID and Y indicates the sequence number of quotes from the corresponding participant.

Table 2.3: Background of interviewees

Group ID	Participant ID	Years of experience	Current role[1]	Gender[2]	Education[3]
1	1	7	D	M	GCS
	2	10	D	M	GCS
	3	7	A	M	GCS
	4	6	A	M	GCS
2	5	11	D	W	GCS
	6	5	D	M	GCS
	7	30	A	M	UOth
	8	24	T	M	GOth
3	9	15	A	M	UCS
	10	5	D	M	GCS
	11	5	A	M	GCS
4	12	13	T	M	UOth
	13	1.5	D	M	GCS
	14	25	P	M	UCS
	15	2.5	D	M	UCS
5	16	4.5	D	M	UOth
	17	9.5	A&P	M	GCS
6	18	3.5	D	M	GCS
	19	2	D	M	UCS
	20	10.5	A	M	GCS
	21	20	A&P	M	GOth
7	22	3	D	M	GCS
	23	13	D	M	GCS
	24	9	D	M	GOth
	25	2.5	D	W	GCS

1. D: developer, A: architect, T: tester, P: product owner 2. M: man, W: woman, none of the participants identified as non-binary.
3. GCS: graduate degree in computer science, UCS: undergraduate degree in computer science, GOth: graduate degree in other science subjects (e.g., electrical engineering, physics and mechanical engineering), UOth: undergraduate degree in other science subjects.

2.2.1.5 Member Checking

Coding is an interpretative process and as such there is always a risk of misinterpretation [164]. In order to reduce this risk, we performed member checking [64], i.e., request interviewees' feedback to improve the accuracy of the derived theory. We emailed each participant two artifacts, the transcript of the interview to remind the participant what was discussed in the interview, and the codes derived from the transcript together with the description of the codes. We encouraged participants to correct us if they disagree with our interpretation, and add new ideas if they would like to do so. We received 20 replies of the participants, of which two required minor changes to the description of the code and two added additional thoughts which did not result in new codes.

2.2.2 Type of Logs (RQ1.1)

The types of execution logs are summarized in Table 2.4.

Table 2.4: Four types of logs (RQ1)

Type	Event log (EL)	Function trace (FT)	Performance data (PD)	Functional data (FD)
Information	Machine event and error message	Order of functions and values of parameters	Duration of software and hardware actions	Business-critical data
Enabled by default	Yes	No	Yes	Yes
Physical layers	Not necessary	Not necessary	Necessary	Not necessary
Quote ID	P20-1	P18-1	P7-1	P8-1

Event logs contain regular events created when a machine action such as initialization has been executed as well as error messages of the systems:"*you will see errors, but also all kinds of events indicating in what state the system is or what phase of execution is being entered*"(P20-1). Developers obtain event logs either from field productions or from in-house test executions. Interviewees use "event log" and "error log" interchangeably.

Function trace contains the details of the program execution. The start and the end of a function call inside components as well as the values of parameters are logged. Compared to event logs, function traces show more details of the execution:"*In the event log you have a higher level view of the system, whereas with component tracing you have a finer level [view] of the system*" (P18-1). Due to performance concerns, function tracing is not enabled by default. Developers can enable it before executing in-house tests. It can be time-consuming to obtain function traces because developers need to set up the simulation environment for test executions and wait for their completion: "*so you have to set up DevBench plans, and they have to run the test, and sync your code. It's already quite some work... sometimes the tests take hours to complete*" (P21-1). To obtain function traces from field productions, developers need to negotiate with customers: "*in order to see this I need tracing from these processes and then you look into if we can at the customer site turn on traces for such a process*" (P9-1). Interviewees use "tracing", "function trace" or "component tracing" interchangeably.

Since the performance (e.g., production throughput) is a key business driver of the machines, **performance data** logs the sequence of function calls at component interface: "*for every component interface, you can specify throughput tag, on entry of a function or an exit of the function, both on the client and on the server side, so you see the start and end points of real function calls* "(P7-1). The performance data logs the duration and sequence of software and hardware actions, showing the *speed* of execution. Obtaining performance data is not trivial because in order to accurately capture the duration of software and hardware actions, the software needs to run on the Testbench: "*You need these Testbenches, which are kind of real machines. For getting access to them you need to arrange it. And you're competing with other people that want to do the same thing. There's only one person who can use the machine at a given moment in time*" (P16-1). Interviewees also refer to "performance data" as "throughput trace".

Functional data logs the business-critical data that represents the functional aspects of the systems: "*it contains details like what is the average heat of wafer*" (P8-1). It can be obtained from field productions and test executions.

RQ1.1 summary

Developers use different types of execution logs that record high-level machine actions, low-level execution details, throughput information as well as business-critical data. Developers need to go through a non-trivial process to obtain the logs because the execution of software for such systems requires hardware to be available or simulated.

2.2.3 Purpose of Log Analysis (RQ1.2)

Table 2.5: The purposes of log analysis and the used logs for those purposes. "#I" indicates the number of interviewees who mention the purpose during interviews.

Purposes	Used logs	#I	Quote ID
Software comprehension			
Familiarizing with existing software	FT	4	P9-2
Reverse-engineering software requirements	FT	1	P3-1
Test development			
Developing test scenarios and code	FT, EL	2	P9-3
Verification and improvement			
Verifying executed behavior vs expected behavior	All	15	P13-1
Performance verification and improvement	PD, FT		
Verifying timing (throughput) performance		3	P16-2
Identifying opportunities of throughput improvement		5	P7-2
Log-quality qualification	All		
Identifying log pollution		1	P19-1
Verifying correctness of the logged information		3	P14-1
Test documentation	FT	2	P16-3
Issue analysis			
Classifying the type of issues	All	3	P21-2
Identifying responsibilities	EL	2	P4-1
Localizing problems	All	12	P1-1
Confirming reproduced field issues	EL, FD, FT	8	P3-2
Identifying root cause	All		
Identifying root cause of field issues		16	P1-2
Identifying root cause of regression test		11	P13-2
Identifying root cause of flaky (test) executions		2	P12-2
Analyzing occurrence and prevalence of issues	EL	2	P22-1
Supporting customers	EL, FD	3	P22-2

We classified 18 purposes into four categories (Table 2.5) and found our results to be complementary to prior studies [44, 224, 436]. Our findings confirm that developers use logs mainly for issue analysis and uncover additional purposes not previously discussed in the literature, such as developing test scenarios and code, reverse-engineering requirements for legacy software, and identifying root causes of flaky executions.

2.2.3.1 Software Comprehension

This category covers two purposes related to comprehending behavior of a system. P3, P9, P14 and P22 use execution logs to complement the source code when familiarizing with the software: "*One of the most important things that you need to understand [is] what the software does, you do that partially based on tracing*" (P9-2). Execution logs also complement the documentation: "*The software is not very well documented. We have to do reverse engineering to get requirements... I can choose to run the current software and enable tracing, and from that tracing, it shows me all the interaction between different components*" (P3-1).

2.2.3.2 Test Development

When developing test cases, developers adopt an incremental approach using logs: "*We analyze the trace... Normally we will start with a very basic scenario of tests. We checked some of the sequence of the essential parts... we continue to extend the test scenario, probably with some pause or stop in the middle and resume it or inject some errors to see if the errors can be handled correctly*" (P9-3).

2.2.3.3 Verification and Improvement

Logs aid developers in verifying and optimizing software performance. Beyond running tests against requirements, developers thoroughly inspect logs to confirm expected software behavior: "*I need to develop some new functionality and we can add some tracing code to the production code then we can look into the tracing whether the behaviors are expected*" (P13-1). In particular, logs help verify that the undesired events do not occur: "*We have a list of events that we say those are not allowed to occur during a regular test, that's where we use the event logs*" (P12-1). In order to achieve high throughput performance, execution logs are also used to verify if actions are finished within their time budget: "*It helps us see how much time a function takes and this throughput tracing is helping us to determine if we are within the time budget for every action that is going on*"(P16-2), and identify if any optimizations can be done: "*Often we get the request to reduce the overall timing, so to do that, you need to know the time it takes and where to and how to reduce that*" (P7-2). Moreover, as part of quality control, developers also check the quality of logs: "*We check whether there is too much logging going on, you know, log pollution*" (P19-1), or correctness of logged information: "*In a project you want to log some events or want to look into some errors, then you need to check if those errors end up in the log"* (P14-1). Since traces and logs represent the behavior of systems, it is also used as part of the test documentation: "*Sometimes we also use this produced trace as content for our test documents that we produced to prove that the change has the intended behavior"* (P16-3).

2.2.3.4 Issue Analysis

Execution logs play an important role in analyzing issues. The issues could be anything that threatens the quality of production, identified by the customers or by in-house test executions. When an issue is reported, as the very first step, developers need to classify it in one of the predefined classes such as functional issues, software issues, or infrastructure issues: "*So it's really first thing what we try to do. It is to classify the issue. This classification helps us to know how to start debugging the code"* (P21-2). By inspecting logs, developers also get a rough idea of which group or person has the expertise to fix the issue: "*We still*

need to find to whom the issue is related, and then start communicating with them to check if our assumption about the issue is correct or not" (P4-1). After the analysis and communication, developers can localize the problems by identifying the suspicious chunk of code that produces error messages shown in event logs: "*The event log gives me an indication that something is going wrong in this component, in this particular file and I cannot understand more from it other than that* "(P1-1). Problem localization helps reduce the scope of the further investigation and answers the question of where the issue occurs. An important step then is to reproduce the field issues in-house with simulation and testing. Based on error messages shown in event logs, developers can confirm that the field issues are correctly reproduced locally: "*We try to mimic the scenario and try to reproduce the error messages as much as we can"* (P3-2). After reproducing the issue, to further identify the root cause of issues (i.e., answer the question of why the issue occurs), more execution details are needed: "*So then I will turn on the tracing for that specific component for details [of the field issue]"* (P1-2). Sometimes, to understand a certain issue better, developers analyze the occurrence rate and the prevalence of the issues: "*when you get some issues with error logs, we can connect to our clients and you can see how many number of times this happened at all the customers... to see if it's really generic or something specific happens at a customer at that point*"(P22-1). There are various kinds of field issues. Sometimes, the parameters (e.g., temperature) shown in functional data are useful to support customers to perform corrections: "*We read the functional data. We try to analyze different kinds of parameters and try to suggest to the customer to run some certain amount of calibration. Because it could be (that) the machine is a bit uncalibrated*" (P22-2). For issues found by testing, developers analyze logs to identify the root cause of regressions: "*Once there are some strange things that we obtain that weren't present in the release before, we need to be pretty sure on what kind of discrepancy is in the error log or the trace"* (P13-2), and flakiness [247]:"*That means they have good runs and bad runs on the same test case. Then we want to know where the instability comes from"* (P12-2).

2.2.3.5 Other Observations

The four types of logs serve different purposes. Event logs show high-level events that help developers map the high-level behavior to components. Function traces provide the low-level execution details of components. Function data are particularly used for issue analysis, while performance data are often used for performance-related purposes. A closer look at Table 2.5 reveals that execution logs are primarily used to analyze issues: indeed, logs are usually the only artifact providing the information about field issues. Applying traditional debuggers to obtain low-level execution information (e.g., variable values) can be infeasible; setting up debuggers for the software executed in the simulation environment requires additional expertise and effort (P24-1). Moreover, debuggers can interfere with timing behaviour and synchronisation between multiple processes: "*What might happen is that you have some timeout, so some processes hanging waiting for the process you are debugging. If he doesn't answer in a short time, it stops. Basically, it throws an error*" (P24-2). This requires developers to log and analyze execution details in function traces to debug such software systems.

We observed differences between software developers. P6, P10 and P20 consider function traces as the last resort when analyzing issues: "*In tracing you can see all the steps within that component, and it can be a lot of data there... if you really cannot see what is wrong then you enable that tracing. But that's really the last resort*" (P6-1). P20 indicated that the use of execution logs also depends on the type of component, e.g., analyzing components responsible for algorithms requires different logging than to control components: "*[the component developed by] my current team is all about calculations, which is not really about control sequence or timing. It just about the numbers. It's a completely different domain. For example, we need that much better [functional] data logging*" (P20-2). Furthermore, the usage of execution logs can be changed with the shift of their roles, e.g., to a product owner: "*I'm more responsible for making sure that the team is executing their work correctly. I myself will not look at logs anymore*" (P21-3).

RQ1.2 summary

Developers rely on logs to obtain low-level execution information for issue analysis that cannot be easily obtained using traditional debuggers. Our findings complement the literature and provide empirical evidences for some additional purposes (e.g., test development).

2.2.4 Information Needs (RQ1.3)

We grouped information needs into five categories as shown in Table 2.6. We observed that developers tend to have common information needs; five types of information are mentioned by more than 10 developers ($>$ 40% interviewees).

2.2.4.1 Context of Issues

As previously discussed, logs assist developers in issue analysis, and many activities, such as identifying responsibilities, necessitate context understanding: "*To be able to create this picture, and later you try to somehow understand based on this picture what went wrong with this run*" (P22-3).

First, developers inspect event logs and functional data to know the settings of the systems:"*we try to look which type of machine, which type of service pack it was, which part of and which type of patch it was*"(P3-3). Second, developers need to understand how the error propagates through the system based on event logs: this requires knowledge of the system architecture and the error handling mechanism. The systems that our interviewees work with employ a Client-Server architecture [280]. ASML implements an error linking mechanism, that is, when an exception occurs in the server component, the server component must notify the client components. Since the same component can play the role of a server towards a group of components, and the role of a client towards other components, it is common that an error propagates from one component to a set of other components that have direct or indirect dependencies on it. Developers inspect logs for records of error propagation to identify the components that might contain the root cause, inferring for which components they need to further inspect low-level details:"*in the error logging it has a tree. The errors are*

Table 2.6: Information needs from execution logs. "#I" the number of interviewees who mention the information need during interviews.

Information needs and sources	#I	Quote ID
Context of issues (EL and FT)		
What are the settings of the machines?	3	P3-3
How does the error propagate?	10	P7-3
At which time point does the error occur? What is the machine doing when the error is raised?	12	P13-3
Data flow and executed sequence (FT)		
In which order are functions being executed?	6	P22-4
What is being executed under current configuration?	4	P1-3
What are the values of variables and how do they flow from one function/module to another?	10	P22-4
State and interaction (FT)		
How do software components interact with each other?	10	P3-4
How does the function sequence change the state of software?	2	P14-2
Timing performance (PD and FT)		
Is there any time gaps between actions?	2	P7-4
Is the software action finished within the time budget?	3	P16-4
Difference between executions (EL, FT and FD)		
What additional errors does the change introduce?	5	P19-2
How do the control sequences of different executions differ?	12	P3-5
How do the functional data of different executions differ?	7	P7-5

linked together, so from the error, I can trace back to the root error and to see when and where actually it happened"(P7-3).

To further understand the behavior of a component when errors occur, developers need the timestamp associated with the error messages, which serves as a linker between high-level information from event logs and low-level details from function traces: "*we can search the timestamp in the software trace to find, let's say, around that moment what had happened*" (P13-3).

2.2.4.2 Data Flow and Executed Sequence

Inspecting the low-level details shown in function traces, developers identify the parts of code that have been executed given a particular setting: "*So a machine to us is sometimes a black box, like you have so many configurations and so many possible inputs, and that changes the output or execution. So to really understand what is being executed under the current configuration [we looked into function traces]*" (P1-3). The order of function execution and the flow of data are important for developers to verify software behavior against their expectations: "*You check two things. If the sequence of the function call is as you expected, given a certain case... and second you check if the generated output which is input for other function, so data moving from one function to another function, is as you expected*" (P22-4).

2.2.4.3 Software State and Interaction

To understand the software system, developers analyze the interactions between components based on the function traces:"*Just to know how the component behaves and what calls went through for example the external boundary of that component and*

how the component reacts with other components"(P3-4). P3, P6, P14, P15 and P22 consolidate the interaction information by means of sequence diagrams.

Developers also analyze how the state of software changes based on function traces. For the components developed with the MDSE approach, each of them consists of multiple state machines that interact with each other. Interactions are realized as function calls and recorded in function traces. Working with such components, developers inspect the interactions between state machines that *change* the states of the system, and compare them with function traces: "*it might go to the wrong path in the state diagram. For example, when it should go back to initialize state, but it's going to the different state and then going to initialize state... so I can look at that trace to see what is the sequence and then look at the model to see if they are matched or there's something wrong*" (P14-2).

2.2.4.4 Timing Performance

Developers analyze throughput traces to improve timing performance: "*Gap is the time between software actions. We can see that there is a gap somewhere in the sequence [in the throughput trace] and then you need to understand where the gap comes from... gaps can be the result of a function calling another function in another task. If the other task is busy doing something else function execution is blocked*"(P7-4), and to verify the timing behavior:"*It helps us see how much time a function takes, and this throughput tracing is helping us to determine if we are within the time budget for every action that is going on*"(P16-4).

2.2.4.5 Differences Between Executions

Developers need the information about the differences between the logs generated from multiple executions in order to, e.g., identify regressions, and understand software changes. This would require one to compare error messages:"*So especially if an error seems to be not consistently appearing, like that caused by some kind of instability, then I want to know which change set most likely introduced it, and then it makes sense to run also older versions of the code to see if it never occurred earlier or not*"(P19-2), function traces: "*Everything is inside one module and then the only thing that we can do is to generate traces in this case, before the change and after the change. And then we say, hey, before the change, the tracing of the external behavior of that component says that it did 12345. But after the change it did 123, and then it jumped into 6, and then 4 and 5 are missing.*"(P3-5), and functional data:"*we will look at this reference output of the calculation and compare it to the output that will be generated by the software after it does the implementation. And if they match each other, we say yes indeed that the calculation and implementation went well*" (P7-5).

Often, developers compare logs generated from multiple executions of one software version to identify the root cause of flaky tests [247]:"*So for those instable test cases, this comparison is also very helpful... so we can compare the bad run with the good run. Then we can know where the instability comes from. Otherwise, sometimes it's really time costly*" (P12-3).

Moreover, the differences between executions can also help identify when machines start deviating from the expected behavior. In machines, produced by ASML wafers move through the production line in batches. The production machines repeatedly perform the same sequence of actions in order to process all elements in the same way. These repeated actions are controlled by sequences

of functions and eventually captured in the function trace. Sometimes, the issue in the machines result in inconsistent actions for these elements. To identify where and when the inconsistency occurs, developers need to identify the differences between the sequences of function calls associated with different elements.

RQ1.3 summary:

Five types of information from logs are mentioned by more than 10 developers. Inspecting the *propagation of errors* is essential to localize the problem. With the *timestamp information*, developers can establish the relations between different types of log. The information about *data flow* and the *interaction of software components* is useful to comprehend the complexity of systems. Particularly, developers need the *differences between executions* for identifying the cause of flaky tests or the deviation from expected behavior.

2.2.5 Tool Support for Log Analysis (RQ1.4)

In this section we discuss the tools developers use, the challenges they are facing when analyzing logs, and the tools they would like to have for log analysis.

2.2.5.1 Tools Used

The interviewed developers are very similar in their choice of tools to analyzing logs. All developers stated that text editors are commonly used. The developers also adopt traditional approaches such as Linux `grep` or their own scripts: "*if I want to do a bit more smarter analysis other than grep and I can do it in Python.*"(P14-3). Although filtering and searching are commonly used to extract information from the log data, there is no joint effort on making a generic tool: "*Now you find a lot of scripts that are used by X by Y by ZXY who don't know each other, but they create the script at a different time*"(P21-4).

When comparing logs generated from different executions, developers either manually inspect the two logs which "*takes a lot of time and it's not really productive*"(P23-1) or use text difference analyzers (e.g., KDiff3, Beyond Compare and Linux `diff`): "*Sometimes I use Beyond Compare for comparing logs. It compares data line by line*"(P2-1).

2.2.5.2 Challenges in Log Analysis

Table 2.7 summarizes the challenges identified.

Log availability and quality In order to enable log analysis, developers first need to collect logs. As mentioned in Section 2.2.2, due to the needs of a physical or simulated environment for software executions, log collection can be a time-consuming process. Particularly, when it comes to log collection from the field, logs are sometimes unavailable due to the performance concerns: "*If you turn on tracing then it slows down the system so heavily that you impact production. It's not something you can do at a customer [site] very easily*"(P9-4); or confidentiality: "*customers are very vulnerable to expose that to us because they*

Table 2.7: Challenges in log analysis."#I" indicates the number of interviewees who mention the challenge during interviews.

Challenges	#I	Quote ID
Log availability and quality		
Absence of logs	8	P9-4, P8-2
Non-standard logging	5	P12-4
Incompleteness of trace	8	P9-5
Presence of noise	18	P8-3
Unreadable format for functions with a lot of parameters	2	P24-3
Missing categorization and overview	3	P13-4
Broken error linking	4	P1-4
Complexity		
Involvement of components from different groups and domains	6	P15-1
Involvement of many state machines	2	P15-2
Presence of concurrency	8	P14-4
Presence of irrelevant differences between logs:	5	
Uninitialized variables		P17-2
Concurrent execution		P11-1, P15-3, P17-3
Timing variation		P17-1
Refactoring		P11-2
New feature implementation		P11-2
Expertise		
Lack of domain knowledge	10	P11-3, P22-5, P22-6
Unfamiliar with code base and software design	9	P7-6, P15-4

don't want that data to become visible to other customers"(P8-2). The quality of logs is also known to influence the developers' ability to perform the analysis efficiently [122, 223, 446]. Indeed, we have the same observations in our context. According to the interviewees, there is no standard way of tracing functions: "*For each software component, they [(i.e., developers)] have their own preference for the format of the tracing. You should be able to read that trace first. Otherwise, it's really not easy*"(P12-4). Where to log and what to log is determined by developers who wrote the code and their peers who analyze the logs might find logging to be excessive (P8-3) or scant (P9-5).

Moreover, working with logs generated from metrology software components comes with a particular challenge. The function calls in such components have numerous parameters recording measurement and modeling data, and subsequently requiring developers to format functions and parameters in logs: "*we have functions with a lot of parameters, and often they're big structures and big arrays and everything is converted into text in trace... Sometimes I really spend time formatting data in a way that I can understand it*"(P24-3).

Given that logs are in the size of gigabytes, and not accompanied by any kind of summary, developers spend a lot of effort and time navigating through them: "*right now all the error messages they are combined or mixed in one file... if the event log can be structured in a better way, then it could improve the efficiency for us to analyze*" (P13-4). Another quality related challenge mentioned by developers is that errors raised by servers are not always linked to their clients due to

implementation bugs of error logging and linking: "*Often what we experience right now is that the error links are broken. And I think this misleads the developer quite a lot* "(P1-4).

Complexity The complexity of a system, including the existence of numerous interacting components, a multidisciplinary environment, and concurrency, can give rise to a multitude of challenges.

Indeed, P15 has indicated that "*You have tracing from multiple software components. They all talk to each other, and that makes it so difficult to understand what was the context of the software before it got there*" (P15-1). For the components that are responsible for process control and implemented using interacting state machines (cf. Section 2.2.4.3), analyzing logs requires tracking the change of states in multiple state machines: "*we have 200 different models. Then you need to check, ok, this model was in this state, and then it calls that model which calls another model and then at some point you're looking at 10 different models and different states, and it's so difficult to understand all the different states.*" (P15-2).

The multidisciplinary character of the software requires developers to analyze the logs capturing the behavior of components from different technical domains: "*sometime maybe the analysis takes days... for example, especially if it is related to other functional clusters [i.e., other functional domains]... I could say that it is the most time-consuming part*" (P5-1).

The machines developed by this company have high competence in processing multiple elements concurrently. This high-level machine requirement is realized by the underlying concurrent software: "*all those process elements they end up in different lists, and then the lists are emptied by different sub-processes... and they all do their things separately, and they synchronize at certain moments. So that makes it difficult, and that is represented and logged in the same trace file in the sequence*" (P14-4). The function trace records function calls from different concurrent executions in a sequential manner, i.e., developers should disentangle interleaved executions.

Complexity does not only hinder comprehension, but also introduces irrelevant differences between logs. Such differences can be introduced by time variation because"*you can see the execution time of functions are sometimes different for different runs*"(P17-1), and uninitialized variables since the values of these variables "*will appear on the trace statement is a random, it's garbage. And if you put this in a tool like Beyond Compare, it will take it as a difference, but in reality, it's not*"(P17-2) Similarly, irrelevant differences can be introduced by concurrency: "*Some events are not necessarily happening in the same order in different executions*"(P11-1), refactoring or implementation of new features: "*You could also see many differences because of refactoring or some development changes we made*"(P11-2). Excluding irrelevant differences requires domain knowledge: "*So if you understand what should be the sequences, then you can basically see, ok, in this case the sequence was flipped, but functionally it's the same*"(P15-3), effort and time: "*more and more preprocessing until you remove the most of them... It costs time. And it can even lead you to wrong conclusions*"(P17-3).

Expertise The systems are not only complex but also multidisciplinary. Working with logs generated from such systems requires domain knowledge

(e.g., how machines expose wafers to the light): "*We can dive into the trace files etc. It is not enough. You have to know what is actually going on here with those traces and what is the component doing*" (P11-3). The analysis is particularly challenging for newcomers: "*Let's say if you have really huge experience in software, but without any ASML knowledge, I would say it is useless... I remember the first year it was really hard for me somehow to understand what's really happening*" (P22-5). Different from newcomers who get lost in the large amount of information in logs, experienced developers such as P14 tend to take a top-down approach: P14 first inspects the interactions between the components that control and coordinate machine actions, and other components. This allows P14 to comprehend how machines were functioning and what functionalities each component have, and to conjecture which parts of machines exhibit faulty behavior. Only then P14 examines execution details for relevant components.

Eight developers stress importance of not only discussion with senior *software* developers as well as collaboration with *functional* developers from other engineering disciplines "*peer working at minimum two, it really helps a lot. Especially when one with nice software skills and the other one with nice functional skills*"(P22-6).

The lack of familiarity with the code base and software design also hinder log understanding: "*you often see a trace of code you never worked on. That's what consumes most of the time*" (P7-6). For example, in order to understand the interactions between software components based on function traces, developers should be familiar with the communication mechanisms between components: "*some of the interactions are based on subscriptions. So you subscribe to event and once that event happened there's a callback. In software tracing you just see there's a handler of the event. If you are not familiar with the structure of the software, you couldn't link that trace [line] with the other component [that gives the callback]*"(P15-4).

2.2.5.3 Expected Tools

Creating multi-level abstraction Developers would like to have a tool that can help them inspect different levels of details from logs: "*On certain levels you can open and close those functions to see what's internally there so that you can maintain a high level overview and details where you need them, instead of only having all the details now, but that's what's happening now, you got a whole bunch of data, and it's all detail*"(P14-5). To provide a "bird's-eye view", the tool can visualize high-level function calls with sequence diagrams, state machines or Gantt charts: "*Usually I end up with drawing the sequence diagrams myself to understand it, but if you could drag and drop traces into a tool and then get a sequence diagram, that would also be nice*" (P9-6). The tool should allow developers to select the level of details they would like to inspect: "*I tend to do that by hand... The problem is that if you generate it, you get everything, not interesting stuff... And then I do it by hand, I just leave that out and only put the interesting sequences in there*" (P17-4). For example, as discussed in Section 2.2.4.3, when dealing with state machine based components, developers inspect the function calls that change the state of state machines. The tool should support developers performing this task by visualizing the sequence diagrams only for these important interactions.

Automatic log comparison Developers would like automatic log comparison tools to provide differences at different levels of details: "*I think presenting all those [comparison] results in a single graphical user interface will be polluting... Maybe we could have maybe multiple options or multiple levels based on what you want to check*" (P18-2). Furthermore, developers envision tools supporting identification of the cause(s) of log differences such as concurrency, refactoring or uninitialized variables.

Providing generic and unified facilities Instead of multiple scripts with (partially) duplicated functionality, developers envision a tool supporting formulation of different queries to different types of logs: "*Such kind of facility would help engineer to start talk to data instead of spending time on parsing*"(P22-6), as well as inspection of the relations between different logs generated from the same execution: "*if we can show different logs in one GUI or one window, then it is easier for us... Currently we just manually go through these logs and find the relationships between logs*"(P2-2). For analyzing errors based on logs, developers expect a knowledge base that stores error patterns identified from historical logs so that the knowledge about errors can be shared across groups.

The tools envisioned should be unified with test and log generation facilities (i.e., DevBench and TestBench) to reduce switching between tools: "*I need to connect to DevBench, fire up my test, then look at each of those files individually, write them to my local files, open the tools like the text editor and then go through each one of them. So basically, if you can unify all of these things at one places, which becomes seamless to go between them, then it becomes super awesome*"(P1-3).

RQ1.4 summary:

Developers mainly use text-based tools to analyze logs. In addition to log quality concerns, *concurrency* and *irrelevant differences between logs* bring additional challenges in log analysis. Developers indicate that they need a tool that creates *multi-level abstraction of executions*, allows them to *compare logs at different levels of abstraction* and provides *generic facilities* that can be shared among developers.

2.3 Replication at Other Companies

In this section, we present our replication study at four companies. This replication study aims at understanding to what extent our findings at ASML are generalizable to other companies (RQ1.5). We start with our methodology (Section 2.3.1) and then report our findings (Section 2.3.2).

2.3.1 Methodology

To understand to what extent our findings at ASML are transferable to other companies, we conducted a replication study. We adopted convenience sampling to recruit four companies. Shull et al. [342] discuss two types of replication study, namely dependent replication and independent replication. The dependent replication relies on the design of the original study as the

basis for the design of the replication, controlling the variations between the original study and the replication. In contrast, the independent replication uses different experimental procedures to reproduce the results. The large number of changing factors make it difficult to interpret the observed differences between the original study and the replication. Hence, dependent replications are recommended to come before independent replications to gain more insights [342]. In this study, we opted for dependent replications by changing the study context while following the same research method (i.e., interviews).

Next, we introduce the design of our interviews, context and participants as well as data collection and analysis.

2.3.1.1 Semi-structured Interviews

We used the same research method adopted in our previous study at ASML (Section 2.2). However, instead of asking open questions only, we asked two types of questions during interviews. First, we asked open questions to trigger in-depth discussion without biasing developers. These questions are the same set of questions that were asked in our previous study at ASML (Table 2.2).

The open questions are then followed by a set of closed questions. The goal of asking closed questions is to validate whether developers from other companies share the experiences of their ASML peers. To this end we compiled the codes that we derived from our previous study at ASML (i.e., codes shown in Tables 2.4–2.7 and codes discussed in Section 2.2.5.3) with a survey-like form: Figure 2.2 shows an example with the closed questions about the type of used logs. During the interview, we first explained the codes to our interviewees and then asked if they share the same experience. We note that we did not include two challenges (i.e., *Broken error linking* and *Involvement of many state machines*) in the validation form because they are specific to the modeling tool and error handling mechanism adopted by ASML. We conducted a pilot study with an industrial embedded engineer to examine whether the questions are well phrased and presented. The engineer suggested that engineers may tend to select all the options about possible tool support (codes discussed in Section 2.2.5.3) especially if their current tools are primitive. Therefore, we dropped the closed questions related to tool suggestions (codes presented in Section 2.2.5.3). Instead, we aim at collecting more ideas about tool support with the open questions related to the used tools, challenges, and tool suggestions.

By asking these two types of questions in this specific order, we aimed to confirm our previous findings while still being able to trigger new insights without biasing interviewees. This replication study was approved by the ethical review board of the Eindhoven University of Technology and the participating companies.

2.3.1.2 Context and Participants

In this study, we involved three companies which develop different types of embedded products and one company which develops code quality checkers. By interviewing both embedded software companies and the company which develops non-embedded products, we would like to get a better idea of the scope of our previous findings.

In this replication study, we aim for reaching a broader audience from several companies. Therefore, we opt for recruiting a smaller number of

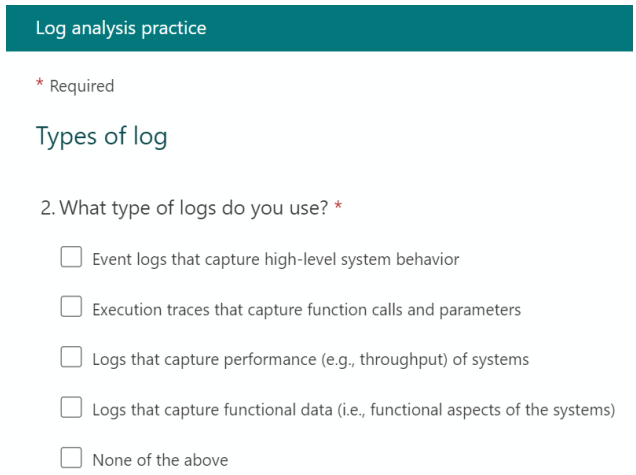


Figure 2.2: Closed question about type of logs used in practice

developers from each company. This replication study is different from our previous in-depth study of a single company (Section 2.2) where a larger number of developers (i.e., 25 developers) are interviewed to ensure that the theoretical saturation is reached. Following the same recruitment procedure as the previous study at ASML, we contacted the managers in the software development division of these companies. We encouraged the managers to recommend six developers to us, while taking into account seniority and diversity of software engineering roles. If the company prefers to provide a smaller number of developers due to the availability of developers, we encouraged the managers to recommend the developers who are experienced with log analysis and knowledgeable of company practices. In total, 14 developers accepted our interviews. Table 2.8 shows the overview of the invited companies and participants.

Company A Company A is a manufacturer of essential components that are required by electronic designs. To produce a high volume of electronic components, the company has built control systems to handle customer orders, logistics and process control. We interviewed six engineers of different seniority levels and roles working on these control systems. The interviewed engineers use the Ada programming language in their development work.

Company B Company B develops various kinds of electronic products which include but are not limit to consumer electronics. In this study, we recruited three engineers. The three engineers work as architect, product owner, and quality engineer, providing different perspectives on the use of logs. The system is mostly developed using C# and C++.

Company C Company C is specialized in developing a certain mechatronic produtc . Two engineers were recruited. One of them is responsible for a

Table 2.8: Participants

Company (product)	ID	Role[4]	Experience[5]	Focus
A (Control systems)	26	D	2	Vision components
	27	D	<1	
	28	A	1.5	Motion control systems
	29	A	5	
	30	D	<1	Data collection platform
	31	A	38	Real-time control systems
B (Electronic products)	32	Q	28	System-wide quality control
	33	A&P	2	Supervisory controller
	34	A&R	10	System-wide design and reliability
C (Consumer electronics)	35	A	3	Data collection platform & test automation
	36	A	24.5	Controller and system interfaces
D (Code quality checker)	37	S	14	System-wide service
	38	D	21	Back-end
	39	D	9	Front-end

4. D: developer, A: architect, P: product owner, S: service engineer, Q: quality engineer, R: reliability engineer
5. Years of experience at the company

data platform that collects data generated from the machines. Meanwhile, the engineer also contributes to the investigation of potential test tooling by studying the state-of-the-art and attending academic conferences. The other engineer is responsible for the software layer for high-level action control and error handling. The system is mostly developed using C# and C++.

Company D Company D is developing code quality checkers that are used in various kinds of software systems. We recruited three engineers. Two of them are responsible for developing the back-end and front-end of the system, respectively. The other engineer is responsible for making sure that products at customer side are working as expected. The front-end of the system is developed using Java and the back-end is developed using Perl.

2.3.1.3 Data Collection and Analysis

To collect and analyze the data, we applied the same method as our study at ASML (Section 2.2.1.4). We collected data by recording audio and making transcripts available. We then applied closed coding, which is a process of identifying and marking interesting information using a pre-established coding scheme [335]. In this study, we used the coding scheme established in our previous study at ASML. We created new codes if the information related to our research questions cannot be labeled with the established codes. We present

our explanation of the derived codes in the following sections. The codes are explained with quotes of developers. We give an ID for each quote to help readers link these codes and the explanations. An ID has a format of PX-Y where PX indicates the participant ID and Y indicates the sequence number of quotes from the corresponding particiapnt.

2.3.2 Generalizability (RQ1.5)

In this section, we present the results of the replication study. As discussed, the study has both exploratory and confirmatory in nature, supported by both open and closed interview questions. By asking these questions, we explore the generalizability of the findings we obtained from ASML with respect to the types of logs (RQ1.1), information needs (RQ1.1.2), challenges (RQ1.3) and tool support (RQ1.4).

2.3.2.1 Types of Logs

As identified in ASML (Section 2.2.2), developers use event logs, function traces, functional data and performance data which are generated separately to support different maintenance activities. Each type of log has its own logging policy and format. In the replication study, we learned that event logs and functional data are commonly generated by companies from ES domain (company A, B and C). Similar to ASML, event logs in these ES companies are generated in a loose text format, while functional data is usually formally defined and formatted through the discussions between software and functional engineers.

However, not all companies generate and use function traces and performance data as ASML. In company A, the functional data and performance data can be generated with an in-house instrumentation technique: "*Developers can put statements in the code where they log certain variables in their ring buffer and that ring buffer is visualized by means of a graphical interface. So you can see how certain, for instance, the X&Y position of a motor or piece of equipment is changing, and also software signals how long certain messages take to get from A to B. All kinds of user defined signals can be in there. There you can see the performance of the machine*"(P29-1). Function traces are not logged at company A due to performance concerns: "*The machines we make are typically very fast. They produce 20 products per second. I estimated (that) each line of log introduces maybe 100 nanoseconds overhead*"(P31-1).

Similar to ASML, company B generates event logs, function traces, and functional data with separate logging formats and mechanisms. The performance data is not logged as a separate type of log. However, according to the interviewed developers from company B, when needed, the duration of actions and events can be inferred from other types of logs (e.g., event logs) based on the recorded timestamps.

The developers from company C shared that the company used to generate one single log file containing different kinds of data in a loose format. But in recent years, the company has separated functional data from the debugging logs. The logged functional data is well formalized and automatically instrumented, and hence can be further analyzed with built-in tools: "*We created the metamodel, so we actually modeled the data that should be logged and how the data relates with each other, and we are trying to define that more accurately by creating a domain specific language and then within that domain specific language we will*

specify what logging we expect. So in that way it is formalized. We have also the logging API that's actually integrated into the embedded software and then used by the software developers"(P35-1). In contrast, the debugging log is manually created by software developers in a loose format: "*There's also no structures, just a string. So, basically all the information that they can come up with, they can just log there*"(P35-2). The free and flexible logging mechanism of this debugging log makes automatic analysis very difficult.

Different from these ES companies where functional data is systematically logged, logging is less formal in company D. The developers at company D manually insert logging statements that capture information, such as events, memory consumption and function invocations, that software developers consider useful. The information is then logged in a single log file at runtime.

RQ1.5-a summary:

Event logs and functional data are commonly generated by embedded software companies. Similar to ASML, the embedded software companies usually formalize functional data for further domain-specific analysis. Not all the embedded software companies log function traces due to performance concerns. Contrary to the separate logging for different kinds of data, company D logs various kinds of useful information into one single file.

2.3.2.2 Purpose of Log Analysis

Figure 2.3 shows the results obtained from the closed questions about the purposes of log analysis. It can be seen that 8 out of 14 purposes have been selected by more than ten developers, and 11 by more than half of the developers, indicating the purposes identified in the study at ASML largely resonate with the developers from other companies. Among these purposes, *problem localization* and *performance improvement* are selected by all the interviewees (=14). Such purposes as *test documentation*, *log-quality qualification*, *developing test scenarios and code*, *identifying responsibilities*, *reverse-engineering requirements*, and *familiarizing with existing software* are mentioned less often (<10). Figure 2.4 shows the distribution of votes for these less frequent purposes over the companies. We can observe the differences between companies; all developers from company B have reported that log quality qualification before delivery is one of the reasons for inspecting logs, while none of the developers from company C recognized it as a common practice. We conjecture that this difference may be due to the different quality control policies implemented at different companies. Furthermore, it can be observed that none of the participants from company D use logs for responsibility identification and code familiarization. As explained by the participants, they can easily perform these tasks by communicating with colleagues because the code base of their system is maintained by a small group of developers.

As seen in ASML (Section 2.2.3.5), logs are widely utilized for issue analysis to recover execution details as breakpoints can introduce synchronization errors, rendering traditional debuggers unsuitable. Our replication study validates the crucial role of logs in issue analysis for embedded systems with stringent

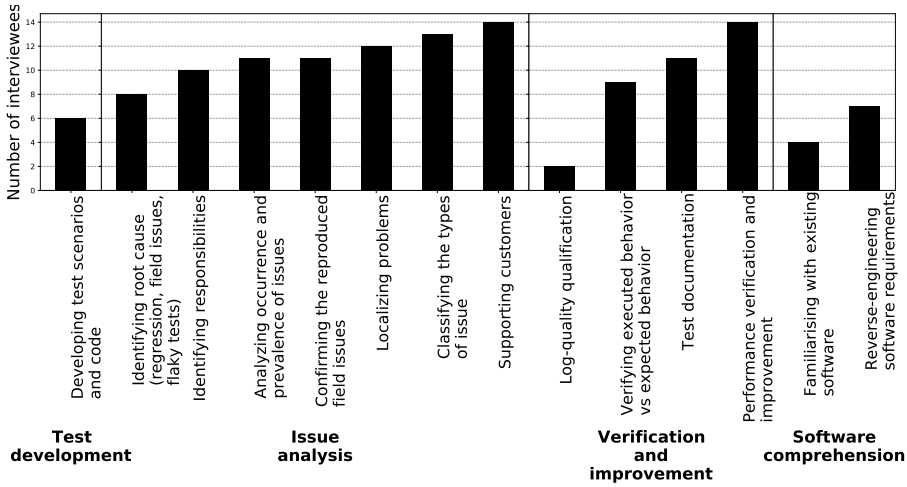


Figure 2.3: Frequency of purposes in log analysis

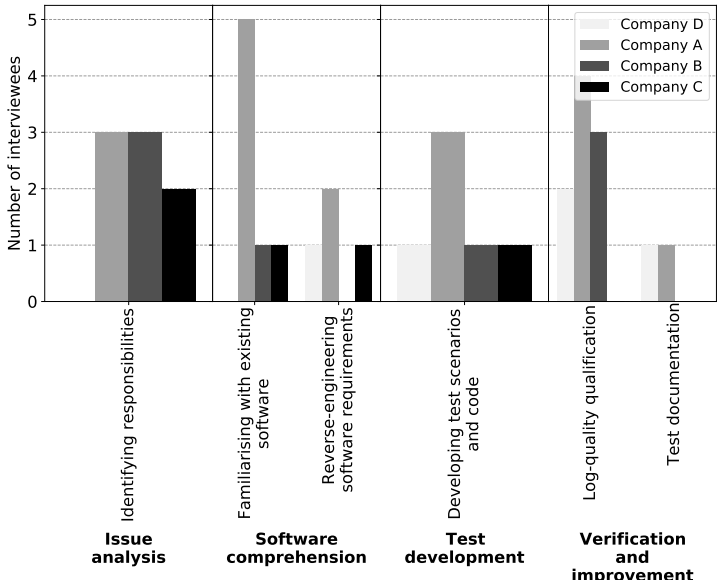


Figure 2.4: Infrequent purposes of log analysis over companies

timing requirements: "*We typically have a watch dog running, so that gives you like a couple of seconds and then the system will automatically reboot, so breakpoint is not an option... not trying to debug with breakpoints but always with logging*"(P36-1).

With the discussion triggered by open questions, we identified three purposes not previously discovered in our study at ASML. Participant P34 shared that Company B has gradually started using logs for more data-driven activities such as **liability analysis**: "*we also started using it to derive some utilization related information, liability related information and also look at obsolescence*

of certain parts. If some old PC is there in the field, you know that that PC doesn't support a newer version of the software or the operating system. Then you should also need to understand how many of those getting obsolete, how should we program the replacement and what is the cost... And then we can also correlate and say how long it's been running. Is it meeting what the vendor is promising in terms of reliability?"(P34-1). As shared by all participants (P32-34) from company B and participant P35 from company C, **use case analysis** is emerging as a purpose of analyzing logs: "*if you have let's say 1000 machines at customers, they would want to see what are now the typical applications that run on the machines. And so that information is gathered by data analysis from the functional data to see what the customers are doing actually and then to be able to improve our products for that*" (P35-3). Additionally, according to participant P35, logs are also used for **testing**: " *in our testing, we write down a couple of steps with synchronization points. So if we have a machine action. We know that if we send a machine action, we first have to wait until the machine is heated up, and then we do the machine action and then maybe the machine needs to cool down, and then we are done. So what you see now with test cases is that we send a command to execute the action. And then in a test case, it says wait until in the logging it shows that the machines are warmed up and then say OK now do the action*"(P35-4).

At company D, we do not see these extra uses. Thus, we speculate that utilizing logs for liability analysis, use case analysis, and testing may be exclusive to embedded software firms with complex machines composed of various hardware components, interacting with operators, and executing actions based on machine state.

RQ1.5-b summary:

The objectives of log analysis from the prior research are commonly adopted by other ES firms. All participants rated problem localization and performance enhancement as important purposes. This confirms the previous finding that a conventional debugger is frequently unsuitable for ES, highlighting the significance of logs in issue resolution. Moreover, we discovered that embedded software companies employ logs for liability analysis, use case analysis, and testing, which were not previously detected.

2.3.2.3 Information Needs

Figure 2.5 shows the results obtained from the closed questions about information needs. Out of 13 information needs, 10 have been voted for by at least 10 developers, indicating that the information needs identified in the previous study (Section 2.2.4) are greatly generalizable to other companies. These most voted information needs are related to *context of issues*, *state and interaction* and *timing performance*. It can also be observed that *configuration of executed systems*, *component interactions* and *duration of software actions against time budget* are voted by all the developers. By discussing the procedure of log inspection, we have also confirmed the findings at ASML that experienced developers often adopt a top-down approach to first get an overview of the execution flow and then drill down to the details.

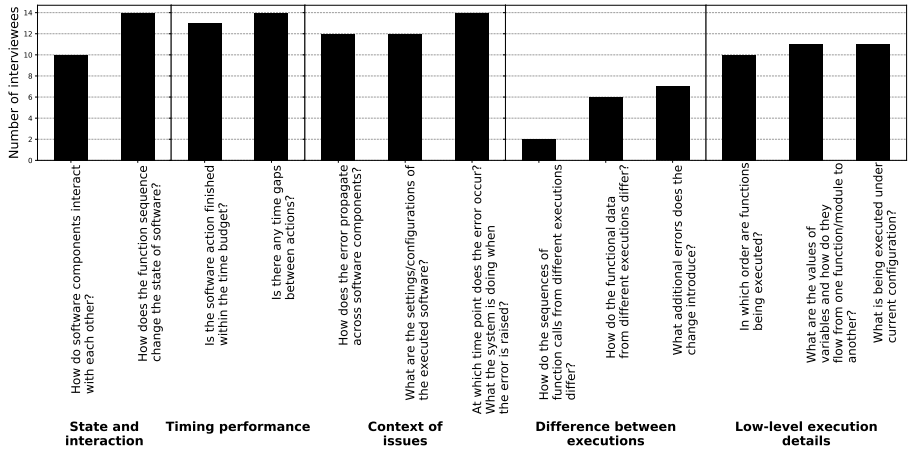


Figure 2.5: Frequency of information needs

We also found that not all developers inspect the *difference between executions* in practice; there are seven developers comparing logs in their practice for investigating regression problems or verifying the correctness of behavioral changes. Particularly, three developers from company A shared that they are not only comparing the sequences of function calls and the values of functional data but also the timing behavior. We identified that obtaining information of **how the timing behaviors from different executions vary** is important for embedded systems used in a fast manufacturing process: "*We have fast machines, so you want actually the variants of the cycle*[6] *also be minimum, because if something is intervening sometimes, and you don't know, that is difficult to oversee. You can measure the cycle. How long does it take? How does that fluctuate? Usually this is very valuable information to see if the responsiveness inside the system is not tampered by something special*"(P31-2).

RQ1.5-c summary:

More than 10 out of 14 interviewees need information related to *context of issues*, *state and interaction* and *timing performance* in their practice. Half of the developers compare logs in their practice for regression investigation and behavioral verification. In addition to the information needs related to *difference between executions* identified in the previous study, we further identify that developers extract the *variants of timing behavior among executions* when comparing logs.

2.3.2.4 Used Tools

The used tools by the interviewees in companies A-D are similar to the used tools identified at ASML. When it comes to information inspection, searching, extraction and comparison, the text-based tools such as a text editor and Linux

[6]cycle time is the time spent on producing an item

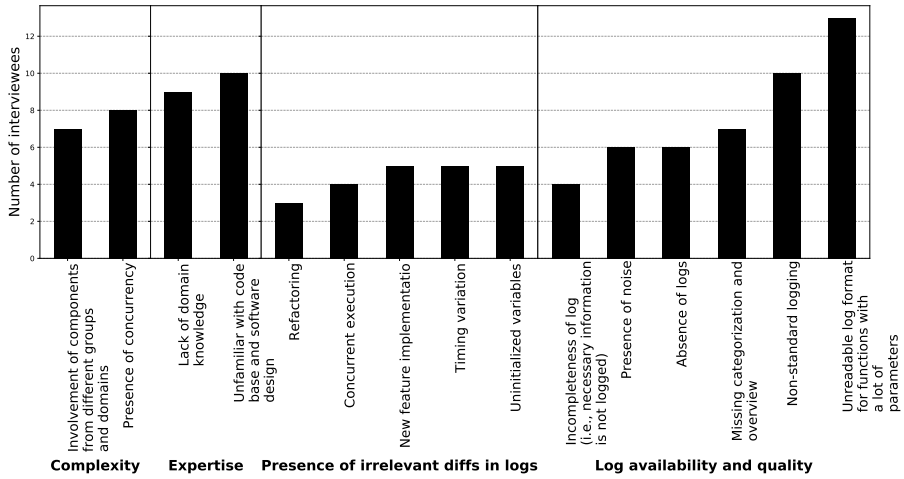


Figure 2.6: Frequency of challenges

grep are most commonly mentioned. In-house tools have also been developed in the companies to inspect logs that capture domain-specific information (e.g., functional and performance logs). For example, company A has developed a tool that can visualize the value of variables over time, which for instance can help developers understand how the temperature of materials in production changes over time. Similarly, company C has also provided tools for developers to analyze the measurements collected in their machines. As explained by developer P35, it is easier to provide tools for functional logs as the collected functional data is usually well-defined against the requirements of systems and the interests of customers. In contrast, it is difficult to provide analysis tools for event logs because these logs are usually loosely formatted as natural language text and cannot be easily parsed automatically. Similar to what we identified at ASML, individual efforts have been made by developers in all these companies to develop customized scripts for parsing and processing these loosely formatted logs. These customized scripts are usually used by an individual developer:"*I made some tooling for myself where I can just visualize the interaction with the controller in sequence diagrams*"(P39-1), or shared in a small group of developers who deal with logs with consistent logging format and similar types of software issues.

RQ1.5-d summary:

Consistent with the observations at ASML, text-based editors and self-made scripts are dominant when it comes to tooling in log analysis practice.

2.3.2.5 Challenges

It can be seen that the challenges recognized by ASML developers (presented in Table 2.7) are not specific to ASML. As shown in Figure 2.6, almost all

interviewees (13 out of 14) consider incompleteness of logs as a challenge in log analysis. According to these interviewees, the challenges of log analysis are often rooted in the challenges of log composition and generation, i.e., if a suitable logging strategy is not applied in the log composition and generation process, then developers have to work with bad quality logs that hinder developing or adopting log analysis tools. Furthermore, the challenges related to expertise and complexity are recognized by more than half of the interviewees (>7). With the open questions, we identified challenges related to *logging trade-off*, *lack of abstraction layer for logging*, and *co-evolution problems in logging*, which were not discussed in our previous study at ASML. We further identify that the coupling between hardware and software in embedded systems has contributed to the complexity of systems.

Logging trade-off. The developers shared several trade-offs in logging. Similar to the challenges reported at ASML, it is often the case that necessary information is missing or incomplete in the log files:"*I encountered a lot of scenarios that there was just no logging available, and I could not conclude what happens here*"(P36-2). A possible solution could be adding more logging statements, but this may lead to an overwhelming amount of information:" *the problem is you never know which part is going to be relevant beforehand, so that's why we put in a lot of logging and then hope when there is a problem that we have captured the correct logging and the correct detail and the correct state. But this can lead to a lot of data and moving around to get to the relevant part*"(P36-3). Moreover, adding more logging code might not be trivial due to project organization: "*you need to formalize request change... it can take years before adding extra logging code due to project organization* (P32-1), long release cycle: "*If you want to insert logging code, you have to go through the software release process* "(P33-1), or performance concern (see a quote from P31 in Section 2.3.2.1). As pointed out by the interviewees, this problem reflects the questions of what-to-log and where-to-log that developers need to answer when logging.

Developers have also reported the trade-off related to logging policy and governance: "*The other challenge, I think, is to have the right balance between complete freedom for the developer to log whatever they want and there's something restricting formalized on the other side*"(P35-5). On the one hand, giving the complete freedom of logging without restriction on the information and format could cause bad log quality such as incomplete logging, non-standard logging format and inconsistent granularity, which results in great difficulties in comprehension and automated analysis. On the other hand, enforcing a strict formalization on logging on what-to-log and where-to-log disallows developers flexibly to add information that they consider useful, which subsequently may also result in missing needed information.

Lack of abstraction layer for logging. Automated logging is considered by developers as one of the ways to avoid mistakes and inconsistencies introduced by manual logging. However, as elaborated by interviewee P29, automated logging requires developers to find a right level of abstraction in the system: "*You may want to automate the generation of log files on the interface level. Then of course, in order to not drown into too many log files at low level interfaces, you have*

to have a certain right abstraction level"(P29-2). The participant further explained that in order to automate logging at the right level of abstraction, the right level of interfaces has to be clearly defined in the architecture of the systems: "*There's no properly defined abstraction level at Company A, where we can do that logging. That is also a challenge because you do not want to do that at very high level or very low level interfaces. You have to find the sweet spot there. We do have interfaces between modules or packages. They're very granular to very low level. Somewhere you have to find a bit of a higher-level to define your interface, so you can trace them without generating too much noise, but still have enough information to follow the internal state of your program*"(P29-3).

Co-evolution problems in logging. Three companies (B, C and D) have shared the challenges regarding logging evolution. Indeed, not only is code that realizes the functionalities of software products evolving, but also the logging code. As a consequence, when logging code is changed, the related entity might require adaptions to preserve its consistency or correctness. For example, as indicated by interviewees P32 and P34, the evolution of logging code affects the maintenance of behavioral patterns that they derived from logs for characterizing known software issues. Developers from company B have been deriving log patterns of the known issues, and storing the log patterns and the corresponding solutions into a knowledge base. The log pattern could be any information that characterizes the issue, such as a sequence of events that manifest the issue. The knowledge database is then shared with developers across different groups and teams for quickly identifying the existence of the known issue in the subsequent versions of software by automatic pattern matching. That is, if an instance of the pattern is detected in the logs generated from the subsequent versions of software, then an issue is found and needs to be resolved with the recommended solution. However, evolution of the software and the logging code threatens to invalidate the patterns derived from a previous version of software: "*Typically these pattern models are affected because of accidental changes in the log statements in the code, or because developers refactored the existing implementation, redesigned components, merged components, or introduced a new feature... these things are quite challenging for the maintenance of the models and patterns*"(P34-2). Moreover, the evolution of logging code also raises the challenges in updating analytical tools: "*data analysis scripts are affected by the change in the logging*" (P35-6), updating customer about new releases: "*We cannot tell our external stakeholders that in this release these are the new logs, these are the existing logs, and these logs we have made obsolete, so stop using them*"(P34-3), and comparing logs: "*so if more logging was added or log statements were changed, then you get these differences. But these are not a reason for different behavior or a failure*"(P35-7).

Coupling between hardware and software. In our previous study (Section 2.2.5.2), we identified that log comparison is difficult in practice due to the presence of differences that are irrelevant to their software engineering tasks. These irrelevant differences could be introduced by concurrent execution, subtle timing variations, refactoring, uninitialized variables and new feature implementation. In the replication study, we learned that the coupling between

hardware and software in embedded systems introduces additional challenges in log comparison. As indicated by the developers from company A, the status of hardware can influence the behavior of software:"*The difference does not always indicate a problem because there is some natural difference in hardware. If you're comparing a log from a machine that was just started between one that has been running for some time, then the motor signals would be different*"(P26-1). Similarly, the variations of software behavior can also influence the behavior of hardware: "*If the software takes too long to do something, then the hardware has to correct it by turning back or stopping... that will change the sequence of function calls from the point on*"(P26-2). As a result, the coupling between software and hardware results in a lot of irrelevant differences in logs generated from different executions.

RQ1.5-e summary:

We identified additional challenges related to the coupling between hardware and software, logging trade-off, lack of abstraction layer for logging, and co-evolution problems in logging.

2.3.2.6 Expected Tools

As a result of the open questions posed during the discussion, we are able to uncover three suggestions for tool development that had not been previously discussed in our interviews at ASML.

Identifying and visualizing dependency between events. As agreed by interviewees from both ASML and companies A-D, comprehending the interleaving of events introduced by concurrency in logs is difficult. Constructing the dependency between events requires a lot of manual efforts : "*It all relies on the mental model. There is no explicit dependencies in logs. You cannot infer the exact temporal dependency. You see a lot of interleaving but do not know the causality between actions*"(P29-4). This is agreed by not only developers of embedded systems but also a web-developer from Company D who indicated the difficulty of grouping events based on their dependencies: "*The difficulty is that sometimes a certain request is not handled by one thread but multiple. And there are many user requests interleaving. So, it is hard to automatically group a certain request and its response for each user*" (P38-1). Not only concurrency, but also the composition of actions introduces the dependency between events: "*Based on our architecture we also have an action administration, so you could imagine for example if you have a machine action then actually a lot of things need to happen. So maybe in the end point the action is decomposed over 500 sub-actions. "* (P35-8). To solve the problems, developers suggest developing a tool that can automatically identify and visualize the dependency between events.

Deriving behavioral fingerprints. As we identified at ASML, developers often manually sketch behavioral models from logs, using them as a vehicle for team communications and software comprehension (Section 2.2.5.3). In the replication study, developers from multiple ES companies consistently suggest that deriving behavioral fingerprints such as behavioral models for known

issues or expected behavior could be very useful for anomaly identification and analysis: "*It's a temporal process that's repeating. It's a cyclical process. So you can easily create a model that we can visualize behavior of the normal execution. So there could be the fingerprint of the process because the same processes are repeatedly occurring*" (P29-5). In fact, as we discussed in Section 2.3.2.5, company B has been deriving patterns for known issues to build a knowledge database that is shared across groups within the company. These patterns serve as fingerprints of known issues. However, developers face the maintenance problem introduced by the evolution of logging code when adopting pattern recognition and matching. To facilitate the use of behavioral fingerprints, there is a need to develop and implement a logging strategy and policy.

Strategic logging. The developers suggest that the process of logging should be defined and governed with company-wide strategies and policies, and tools are required to facilitate the following activities:

(a) *Creating parsable logs*. As observed, these companies are still widely adopting a conventional logging approach [154] where logs are loosely formatted. Loosely formatted logs cannot be easily parsed automatically, and are subsequently hard to be processed and analyzed by automatic tools. Indeed, as indicated by the interviewees, it is currently difficult to create generic parsers that can be used by different groups. Therefore, a better approach could be formatting the contents of log messages in the logging code to generate parsable logs. For large-scale companies, the conventional logging approaches and libraries have often been used for decades in a large code base. Hence, it requires tremendous efforts to manually format all the logging code or migrate to a new logging mechanism: "*our logging library is flexible enough that people have used it in different ways and there is no one pattern to look at how the logs are written in the code*"(P34-4). Therefore, interviewees expect tools that can automate the re-engineering activity. As interviewee P34 suggested, the re-engineering activity may require applying code analysis techniques (e.g., static analysis) to recognize the logged information (e.g., parameters in each event) and migrating an old logging library to a new one automatically.

(b) *Identifying logging changes*. To cope with the problems during the evolution of logging code, identifying the changes that developers made to logging code becomes essential. Developers expect a tool that can identify the changes in logging code and generate an overview of the made changes: "*Did you accidentally remove the entire logging? Did you just change the meaning of the log itself? So every bit of information in the log should be checked. And that should be checked at the development time itself. So when you deliver your code, you should be able to quickly check and say you are breaking an existing log. They should be able to go back and, revert the change and provide justification if they want to go ahead with changes.* "(P34-5). The generated overview of logging changes can enable further analysis, such as interpreting the differences in the logs generated from two versions of software.

(c) *Impact analysis of logging changes*. As identified, developers face co-evolution problems in logging. The entities such as analytical tools, behavioral fingerprints, and knowledge databases are impacted by the changes in logging code. In order to evolve these entities to preserve their consistency and correctness, developers expect a tool that can analyze the dependency between the changed logging code and these log-dependent entities, suggesting the required adaptions to developers.

RQ1.5-f summary:

In order to tackle the challenges, the interviewees suggested tools that can identify dependencies between events in logs, derive behavioral fingerprints from logs and support strategic logging. In particular, tools are suggested to support the adaptions of log-dependent entities that are affected by the evolution of logging code.

2.4 Result Synthesis

In this section, we synthesize the two studies presented in Sections 2.2 and 2.3. We first discuss the main scenarios of software logging identified in these two studies. As presented in Section 2.3, the findings about types of logs, and purposes, information needs, and challenges of log analysis identified in ASML are applicable in other companies. However, we also observe that some contextual factors (e.g., different types of components and programming languages) may lead to variations of logging and log analysis practices (e.g., using a certain type of logs more often). In this study, with a limited number of interviewees from different development groups and companies, we do not focus on the exploration of these influencing factors. Instead, we report our observations and formulate our hypotheses that can further be validated later by a survey or repository mining study.

2.4.1 Main Scenarios of Software Logging

By synthesizing the data collected from 39 engineers, we observe a main scenario of logging. As we discussed in Sections 2.2.3 and 2.3.2.2, *Issue analysis* is the main purpose for which engineers analyze logs. We observe that this purpose often appears with information needs *Context of issues*, *State and interaction*, *Data flow and executed sequence* and *Difference between executions*. The co-occurrence indicates that these types of information are most essential for analyzing software issues. This is aligned with the general procedure that developers often follow to analyze software issues. Understanding the context of issues is an important step to recognize the symptoms and localize the issues (i.e., identifying the suspicious components). This involves the inspection of error propagation shown in the event logs with a top-down approach: "*so usually we start with the error message that is important on the highest layer... It might be that always something went wrong there and there is no lower layer involved.*

And if it's indeed going down one layer... and then we go to hardware layer, what's going on there" (P11-4). As indicated by developers, understanding the context of issues requires developers to have a very broad knowledge of systems and their architecture. Once the suspicious layer and components in the layer are identified, the interactions between these components (shown in function traces) are inspected by filtering function traces on the function calls across components. This activity helps developers inspect the external behavior of components, requiring them to have a mental model of how a cluster of components interacts with each together. If the interactions between components deviate from the expected high-level system behavior, developers further dive into the internal behavior of components by inspecting *Data flow and executed sequence* shown in function traces and functional data. We can observe that the architectural knowledge of systems plays an essential role in scoping and localizing the issue for such complex and heterogeneous systems, while knowledge of low-level code behavior is essential for identifying the root cause of issues.

Frequently, developers, especially those who are new to the companies, may lack the necessary architectural and coding expertise to conduct a thorough root cause analysis. To pinpoint the location and underlying cause of problems, comparing the logs generated during failed and successful executions is an effective method. This comparison is often performed for different types of logs, as discussed in Section 2.2.4. For example, by comparing the function sequences that show component interactions of two executions, one can identify if the issue is caused by the violations of interaction protocols between components. The comparison practice, however, is challenging due to the large number of irrelevant differences returned by text-based tools (as discussed in sections 2.2.5.3 and 2.3.2.6). Log comparison is particularly effective for analyzing the root cause of flakiness. In this case, logs are generated from multiple executions of one software version (see quote P12-4). The comparison result between them does not contain the differences caused by software modifications, but only the differences that are likely to uncover the non-deterministic runtime behavior.

Apart from log comparison, junior developers leverage additional information to complement their partial knowledge of the domain and architecture. As discussed in Section 2.2.5.2, peer-working with functional engineers is useful to interpret log information. Moreover, correlating the log information with the development activities can help them identify the cause and effect. The interviewees often check the software repositories to identify recent code changes that may introduce the issues. Being aware of the development activities of other groups that are responsible for the interfaced components is also useful for developers to quickly identify the possible violations of the interaction protocol.

It is worth noting that, as the interviewees indicated, there is no fixed way to analyze software issues. Depending on the types of issues, the pre-knowledge developers have about the issues, and the type of software components which cause the issues, the procedure and needed information may vary.

2.4.2 Contextual Factors in Logging Practice

Based on our interviews with the developers from different companies, domains and development groups within a company, we hypothesize that types of systems, types of components, architecture and complexity of systems, and used programming languages are contextual factors that may influence developers' practices. In this subsection, we provide observed evidence that support this hypothesis, which should be further explored and validated with a systematic empirical approach.

2.4.2.1 Types of Systems

To explore the scope of our findings, we involved four embedded software companies (i.e., ASML and companies A-C) and one company that develops general applications (i.e., company D). Since no new insights about the types of logs, purposes of log analysis, information needs, challenges and expected tool support are identified from the interviews with company D, we conjecture that most of our findings from embedded software companies are not specific to the context of embedded systems. However, we expect that companies that develop different types of systems may perceive the severity of these challenges differently. As observed, on the one hand, log analysis is essential because it is often the only way to inspect the internal states and execution details of embedded systems. Logs are heavily used by developers for such systems because of the difficulties of using a traditional debugger. This observation concurs with the theory of probe effects—traditional debuggers are ill-suited for concurrent systems because the injection of breakpoints (i.e., delays) may change the system behavior [124]. On the other hand, logging statements introduce overhead that may violate the critical timing requirements of embedded systems. On top of that, it can be a very iterative, and resource and time consuming process to execute the systems and collect logs (as discussed in Section 2.2.2). It is therefore considered by most interviewees a challenging task to log minimal but sufficient information for embedded systems. This paradoxical observation emphasizes the importance of effective logging techniques and guidelines for embedded systems. In contrast, developers from company D stress less concern about performance overhead but more about identifying relevant information from a large amount of log information.

2.4.2.2 Types of Components

An embedded system is composed of many types of components. Often, different types of components have different logging strategies and for analyzing the issues caused by them a different log analysis practice is followed. For example, P20 has worked in two different groups of the company. According to P20, different types of components require different testing strategies to expose issues, use different logging approaches, and rely on different kinds of log information, and use different logging approaches and strategies. The previous group is responsible for the control actions of the machines, while the current group is responsible for the algorithmic applications (e.g., calibration algorithms) running on the machines: "*it is a completely different domain with*

different problems. Their way of testing is very different. They usually need some online system tests where you actually expose wafers in order to find problems in testing. And in my current team it's all about calculations, which is not really about asynchronicity or timing. It is just about the numbers...we tried to set things up as like small modules without any external dependencies, like standalone stuff and that does allow us to make more unit tests"(P20-3). Due to the differences, the previous group relies on event logs, funtion traces and performance data which show action synchronization and timing while the current group relies on functional data produced by the calculations and measurements (see quote P20-2). The properties and requirements of components also influence how much logging a component allows without impacting the overall performance of machines.

2.4.2.3 Architecture and Complexity

Different embedded systems may have different architectural designs, and exhibit different levels of complexity. P29, who is currently working in Company A, has worked at ASML before. The interviewee shares views about the systems developed by these two companies: "*The architecture of ASML systems definitely makes tracing easier because they have a natural interface. They explicitly defined their interfaces for components, and that makes a very natural boundary for tracing... But ASML systems are much bigger. So it's easier for our company in that sense because our systems are less complicated*" (P29-6). Indeed, ASML defines the interface between components and traces the function calls at the interface, which allows developers to inspect the interactions between components. In contrast, Company A has interfaces at a more granular level, which may generate too many details (see quote P29-3). This comparison shows that architectural design and complexity of systems are important factors that contribute to the difficulties of software logging practices. It emphasizes the importance of taking logging into account at the design phase of systems, and properly defining the abstraction level for software logging.

2.4.2.4 Programming Languages

Embedded systems can be implemented by different programming languages, which may lead to different software logging practices. P30 from Company A, who uses the Ada programming language, shares that the ability of specifying constraints in the language may lead to less logging: " *so, I was grown up with C and C++. But Ada is way better in its type system. Now you can define all the constraints on the type, and then you can always be sure that your type is correctly constructed and then if you set these pre-conditions or post-conditions for your function correctly, then there is no need to log these parameters and functions in my perception*"(P30-1). We observed similar ideas in ASML where a state machine modeling language is adopted to verify the correctness of software behavior. P14, who adopted this modeling language in their project, expects the verification will reduce the needs for software logs: "*Maybe the question is how relevant are event logs and traces? Because it's expected that there will be fewer issues, in the sense that it prevents the developer from adding logic errors in software, but we are not sure yet if that is indeed the case. Let's say, at least from a practical point of view, we need to live with it for a while and see what happens*"(P14-6). This hypothesis is

supported by P3 and P7, who have used the modeling language for a while: "*we use state machine models and these state machine models are formally verified. We are let's say 95% sure that the problem is not in the generated code*"(P3-6).

2.5 Discussion

There are two lines of work in the field of software logging. One line of work is empirical studies which aim to help researchers understand developers' practices, gaining design knowledge for the development of techniques that can solve real-world problems. This line of work collects empirical evidence by mining software repositories or surveying developers. Another line of work focuses on proposing techniques that solve a certain problem in software logging. Our work, collecting the perceptions from industrial developers, contributes to the first line of software logging research.

We study the relevant empirical studies about software logging practices that appear in several literature studies about software logging [77, 132, 154]. In particular, we compare our work against recent empirical studies on software logging practices. We compare our work against the relevant empirical work in three ways. First, we summarize the context and topic of the relevant studies and discuss the complementary nature of our work to the existing body of research (Section 2.5.1). Second, we provide the refined taxonomy obtained from our work and compare the taxonomy with relevant work (Section 2.5.2). Third, we highlight the main findings of our work and discuss their alignment with relevant empirical work (section 2.5.3 and 2.5.4).

Finally, we discuss the recent research about log analysis techniques at ASML (Section 2.5.5) since the completion of our exploratory study at ASML. These research studies confirm the usefulness of our findings and implications for researchers and tool builders. Furthermore, they demonstrate how the research outcome of our study can be transferred by other researchers to solve real-world problems.

2.5.1 Topic and Context of Relevant Work

Table 2.9 summarizes the research approach, type of research, context, and type of application. Our work is complementary to existing literature.

First, our work focuses on a different phase of software logging practices. Chen et al. [77] conduct a systematic literature review on software instrumentation and divide software logging into two main phases: log instrumentation and log management. Log instrumentation refers to the steps of the integration of a logging library and the composition of logging code. Log management refers to the steps where logs are generated, collected and used for the analysis of system behavior. The majority of existing work focuses on the phase of log instrumentation (e.g., where-to-log, what-to-log and how-to-log). Our study focuses on log management phase, where log collection and analysis are involved to achieve developers' intentions with logging. We focus on this phase because we believe that by understanding the challenges that developers (as end users) face in log analysis, we can better recognize the problems that lie in the phase of log instrumentation (as presented in Section 2.2.5.3 and 2.3.2.5)

Table 2.9: Literature about logging practices. "-" indicates that the information is unspecified in the corresponding paper.

Reference	Topic	Method	Context	Domain	Language
Yuan et al. [432]	Log prevalence and modification	A mining study of four projects	OSS	Various	-
Chen et al. [76]	Log prevalence and modification	A mining study of 21 Apache projects	OSS	Various	Java, C and C++
Pecchia et al. [295]	Logging ponits, purposes and challenges	A mining of three systems, inspection of 2.3 millions log entries and query feedback from the development team	Industry	Critical system	C and C++
Li et al. [224]	Benefit and cost of logging	A survey of 66 developers and a case study of 223 logging-related issue reports.	OSS	-	-
Rong et al. [322]	Logging intentions and concerns	A series of interviews and a mining study of three projects	Industry	Big-data technology	Java
Rong et al. [321]	Consistency of logging practice	A mining study of 28 projects	OSS	Various	Java
Harty et al. [153]	Logging prevalence and information	A mining study of 57 projects	OSS	Mobile App	Java
Fu et al. [122]	Logging points	A mining study of two systems and a questionnaire survey with 54 developers	Industry	Cloud application	C#
Zeng et al. [436]	Logging purposes	A mining study of 1,444 projects and an email interview	OSS	Mobile App	Java
Barik et al. [44]	Logging purposes and challenges	Interviews with 28 software engineers, and a quantitative survey of 1,823 respondents	Industry	Cloud application	-
Kabinna et al. [184]	Impact of logging library migration	A mining study of 233 Apache projects	OSS	Various	Java
Gholamian et al. [133]	Logging overhead	An experimental study on seven Spark benchmarks	OSS	Distributed system	Java
Kabinna et al. [185]	Logging modification and stability	A mining study of four projects	OSS	Various	Java
Shang et al. [340]	Information needs of users	A qualitative analysis of 15 email inquiries and 73 inquiries from web search about different log lines	OSS	Distributed system	-

and identify the techniques that can aid developers in the log management (as presented in Section 2.2.5.3 and 2.3.2.6).

Second, our study contributes to the understanding of log analysis practices for embedded systems. We can see from Table 2.9 that previous studies are conducted for various types of systems (e.g., cloud applications and Mobile App). As identified by Gholamian et al. [132], who conduct a comprehensive systematic review on the subject of software logging, including practices and analysis techniques, domain-specific studies about software logging practices are needed because different types of systems may require different practices (e.g., recording different types of information).

Table 2.10: Refined taxonomy for log analysis. "∗" indicates the codes that are newly discovered in the replication study. "Ref./New" indicates the reference of related literature that is aligned with the corresponding code or a new code that has not been observed in prior work.

Types of logs	**Ref./New**	**Quote ID**
Event log	[295]	P20-1
Function trace	[295]	P18-1
Performance data	New	P7-1
Functional data	[295]	P8-1
Purposes	**Ref./new**	**Quote ID**
Software comprehension		
Familiarizing with existing software	[224]	P9-2
Reverse-engineering software requirements	New	P3-1
Test development		
Developing test scenarios and code	New	P9-3
Verification and improvement		
Verifying executed behavior vs expected behavior	[44, 224]	P13-1
Performance verification and improvement		
Verifying timing (throughput) performance	[436]	P16-2
Identifying opportunities of throughput improvement	[436]	P7-2
Log-quality qualification		
Identifying log pollution	New	P19-1
Verifying correctness of the logged information	New	P14-1
Test documentation	New	P16-3
Testing*	[44]	P35-4
Use case analysis*	[44]	P35-3
Liability analysis*	New	P34-1

Continued on next page

Table 2.10: Refined taxonomy for log analysis. "∗" indicates the codes that are newly discovered in the replication study. "Ref./New" indicates the reference of related literature that is aligned with the corresponding code or a new code that has not been observed in prior work. (Continued)

Issue analysis		
Classifying the type of issues	New	P21-2
Identifying responsibilities	[224]	P4-1
Localizing problems	[436]	P1-1
Confirming reproduced field issues	New	P3-2
Identifying root cause		
Identifying root cause of field issues	[44, 76, 224, 322, 436]	P1-2
Identifying root cause of test issues	[44, 224]	P13-2
Identifying root cause of flaky (test) executions	New	P12-2
Analyzing occurrence and prevalence of issues	New	P22-1
Supporting customers	[44, 224]	P22-2
Information needs	**Ref./new**	**Quote ID**
Context of issues		
What are the settings of the machines?	New	P3-3
How does the error propagate?	New	P7-3
At which time point does the error occur? What is the machine doing when the error is raised?	New	P13-3
Data flow and executed sequence		
In which order are functions being executed?	New	P22-4
What is being executed under current configuration?	New	P1-3
What are the values of variables, and how do they flow from one function/module to another?	New	P22-4
State and interaction		
How do software components interact with each other?	New	P3-4
How does function sequence change the state of software?	New	P14-2
Timing performance		
Is there any time gaps between actions?	New	P7-4
Is the software action finished within the time budget?	New	P16-4
Difference between executions		
What additional errors does the change introduce?	New	P19-2
How do control sequences of different executions differ?	New	P3-5

Continued on next page

Table 2.10: Refined taxonomy for log analysis. "∗" indicates the codes that are newly discovered in the replication study. "Ref./New" indicates the reference of related literature that is aligned with the corresponding code or a new code that has not been observed in prior work. (Continued)

How do functional data of different executions differ?	New	P7-5
How do timing behavior of different executions differ?*	New	P31-2
Challenges	**Ref./new**	**Quote ID**
Log availability and quality		
Absence of logs	[224]	P9-4,P8-2
Non-standard logging	[295, 321]	P12-4
Incompleteness of log	[224]	P9-5
Presence of noise	[224]	P8-3
Unreadable format for functions with a lot of parameters	New	P24-3
Missing categorization and overview	New	P13-4
Broken error linking	New	P1-4
Complexity		
Involvement of components from different groups and domains	[44]	P15-1
Presence of concurrency	New	P15-2
Presence of various kinds of differences between logs caused by:		
Uninitialized variables	New	P17-2
Concurrent execution	[146]	P11-1,P15-3,P17-3
Timing variation	[146]	P17-1
Refactoring	[146]	P11-2
New feature implementation	[146]	P11-2
*Coupling between software and hardware**	New	P26-1, P26-2
*Change of logging code**	New	P35-7
Expertise		
Lack of domain knowledge	New	P11-2,P22-5,P22-6
Unfamiliar with code base and software design	New	P7-6,P15-4
Logging*		
Logging trade-off*	[224, 295]	P35-5
Lack of abstraction layer for logging*	New	P29-2,P29-3
Co-evolution problems in logging*	New	P34-2,P35-6,P34-3,P35-7

Continued on next page

Table 2.10: Refined taxonomy for log analysis. "∗" indicates the codes that are newly discovered in the replication study. "Ref./New" indicates the reference of related literature that is aligned with the corresponding code or a new code that has not been observed in prior work. (Continued)

Tool support	Ref./new	Quote ID
Creating multi-level abstraction	New	P14-5,P9-6,P17-4
Automatic log comparison	[146]	P18-2
Providing generic and unified facilities	New	P2-2,P1-3
Identifying and visualizing dependency between events*	New	P29-4,P38-1,P35-8
Deriving behavioral fingerprint*	New	P29-5
Strategic logging*	[224, 295, 322]	P34-4,P34-5

2.5.2 Refined Taxonomy for Log Analysis

Our study overlaps with existing work in several areas, such as logging purposes and challenges. We provide a detailed discussion of this alignment in Table 2.9. We refined the taxonomy presented in Section 2.2 based on the results of the replication study in Section 2.3. The refined taxonomy, shown in Table 2.10, includes three additional purposes related to verification and improvement, one information need related to differences between executions, two challenges related to various kinds of log differences, three challenges related to logging, and three suggestions for tool support.

Our study contributes new codes to the existing empirical literature on logging practices, including types of logs, purposes, information needs, challenges, and tool support. Table 2.11 summarizes our main findings and their alignment with existing literature. Our findings cover both log instrumentation and log management phases. We will discuss the findings in more detail next.

Table 2.11: Major findings and implications of our study. In the bracket in implication column, "R" indicates implications for researchers, "T" for tool builders, and "P" for practitioners.

Log instrumentation	Literature	Implication
Logging in embedded systems, on one hand, often suffers from the probe effect if logging is excessive, and on the other hand, is essential for issue analysis because traditional debuggers are often not feasible.	Logging overhead has been discussed by many empirical studies [144, 154]. Our study emphasizes the criticality of the problem in the context of embedded systems.	Conducting studies of logging practice in embedded systems: 1) the impact of logging on different parts of embedded systems (R), and 2) to what extent the current logging practice satisfies embedded developers' information needs (R).
Shifting logging decisions to design time of systems is championed by different roles of embedded engineers who all experience missing essential log information for their tasks. Particularly, systems need to be well-architected to support logging at a suitable level of abstraction.	The idea of making logging decisions in the design phases is aligned with a suggestion from Rong et al. [322]. Based on the perception of embedded engineers with different roles, our study stresses that making logging decisions at design phase with stakeholders is particularly essential in the embedded domain which is multidisciplinary by nature.	Developing a suitable logging strategy at the early phase of system development with stakeholders of logs (P).
Log management	**Literature**	**Implication**
Multiple types of logs are used by developers to extract information most related to context of issues, state and interaction and timing performance in their practice.	The use of multiple types of log information is also observed in the studies by Pecchia et al. [295], Harty et al. [153], and Shang et al. [340] performed in different contexts. Our study shows the importance of timing information in the context of embedded systems.	Linking multiple types of logs to obtain a comprehensive picture of systems (T).
Log comparison is practiced by developers for various activities such as investigation of software regression and flakiness. However, comparing logs is difficult due to the presence of many irrelevant differences.	The need for log comparison has been also identified in Microsoft [44] and Google [146]. In our study, we detail the type of information developers compare and the challenges they face in practice.	Identifying the sources of log differences to support maintenance tasks (T).
Log comprehension is often hindered by the lack of code and domain knowledge, and the presence of concurrent executions of systems. Experienced developers tend to adopt a top-down inspection approach, and obtain abstraction by sketching behavioral models based on logs.	Little empirical study has reported findings about log comprehension.	1) Creating multi-level abstraction of executions to support log inspection and comparison (T), and 2) augmenting logs with additional information (T&R).
The problem of co-evolution between logs and log-dependent entities occurs due to the evolution of logging code.	Many studies show that logging code is modified by developers [76, 184, 432]. However, no previous study provides evidence of co-evolution problems.	Supporting co-evolution in log analysis (R)
Manual analysis with text editors are a common practice in the studied companies.	Manual analysis with text editors has been observed in other companies [44, 292].	Identifying gaps between the state-of-practice and state-of-the-art of log instrumentation and log management (R).

2.5.3 Log Instrumentation

Among the challenges identified in Table 2.10, seven are related to log availability and quality. The interviewed companies largely follow the method of conventional logging [77] that gives developers a lot of freedom to manually place logging statements scattering across the code base and generates free-formed logs, which subsequently introduces difficulties in the analysis steps. This observation triggers the difficult question of to what extent and how logging policies should be enforced. Indeed, as discussed by the interviewed developers, when logging software systems, developers need to make several trade-offs. A lot of effort has been made in the research community to study such questions as where to log [122, 223], what to log [446], how to log [76] and how to use logs [148]; and such challenges as absence of logs [224], non-standard logging [295], and presence of noise and incomplete logging [224].

2.5.3.1 Logging in Embedded Systems

As we discussed in Section 2.4, the major challenges of software logging faced by developers from Company D and Companies A-C are different. Performance overhead remains the major concern when it comes to logging for embedded systems. Indeed, it has been shown that different types of systems have different logging practices. Zeng et al. [436] find that logging in mobile apps is less pervasive and modified than server and desktop applications. By comparing the app performance between enabling and disabling logging, they find that logging can induce a statistically significant performance overhead. Another example can be seen in the mining study conducted by Gholamian et al. [133] where the impact of different logging granularities are evaluated in the context of distributed systems. As a result, they observe on average 8.01% and 268X overhead in the execution time and storage when the trace log level (e.g., the more detailed logging level) is enabled versus the info level (e.g., the coarser logging level).

There is little quantitative study (e.g., repository mining) on the logging practice for embedded systems. The relevant questions remain unanswered: what developers actually log in their systems, to what extent logging impacts the performance of such systems, and whether the logged information satisfies developers' information needs that are identified in this study. Getting insights into these questions can help researchers propose techniques and guidelines to resolve the logging trade-off for embedded systems. Our study further suggests that the type of components and programming languages should be taken into account while conducting such studies.

As observed in our study, different types of components in an embedded system and different used programming languages lead to different logging needs (see Section 2.4.2). It is conjectured in the literature that OSS projects with a different programming language may have different logging practices: Chen and Jang [76] conducted a replication study with Java projects and obtained quite different results from the original study which was conducted with C/C++ projects by Yuan et al. [432]. With the evidence shown in our study and the

literature, we suggest researchers to deepen the understanding of software logging for embedded systems by taking these contextual factors into account.

2.5.3.2 Logging Decisions at Design Phase

The interviewees suggest that the design and implementation of logging approaches should be considered at the design phase of system development. This suggestion is aligned with a suggestion from Rong et al. [323]. Moreover, missing logging guidelines that systematize the logging process have been reported by existing studies [295, 323]. Indeed, this follows the conventional wisdom in data-intensive activities: garbage in, garbage out. We compile two suggestions for practitioners about logging practices based on our observations in this study.

By nature, embedded systems are developed and maintained by multidisciplinary groups of engineers. As observed, not only software developers but also engineers who are responsible for function design, customer service and quality assurance also use logs in their daily work. These engineers with different roles have experienced difficulties in log analysis, such as information missing in logs. This observation emphasizes the need for defining what-to-log and where-to-log with the stakeholders who use logs for their engineering tasks. Furthermore, a set of terms and their semantics should be defined through discussions to represent the domain-specific concepts. Furthermore, consistent with a suggestion provided by literature [77], the developers suggest considering automatic logging at certain locations (e.g., interfaces of software modules) following designated rules. As further discussed by the interviewees, in order to automatically instrument software with an appropriate and consistent granularity, the systems need to be well-architected with an appropriate level of abstraction. This idea concurs with the widely accepted software engineering practice that various stakeholders should be actively involved in requirement engineering activities [272, 292]. That is, the requirements of logging should be considered as system requirements which are discussed at the phase of system design with stakeholders. Particularly, the heterogeneous nature of systems and logging needs also lead to questions about how to standardize software logs generated from components using different logging libraries in different programming languages. Moreover, to ensure the quality of logs, methods and practices such as automatic checkers or code review should be adopted to identify and govern the modification of logging code.

2.5.4 Log Management

In addition to the findings about log instrumentation, we have several findings related to the management and analysis of logs.

2.5.4.1 Multiple types of logs

As shown in Table 2.10, we observed that developers use four types of execution logs and five categories of log information in their embedded software engineering practice. Pecchia et al. [295] analyzed the codebase of an industrial

critical system and found that developers logged the value of critical variables, invocations of functions, and occurrence of events of interest, which corresponds to the event logs, function traces and functional data identified in our study. Harty et al. [153] identified four types of information are usually logged in Mobile Apps: business events, user interface events, failures and/or unexpected situations, and other information. By analyzing 15 email inquires and 73 inquiries from web searches for three open source systems, Shang et al. [340] identified five types of information (i.e., meaning, cause, context, solution and impact) that *users* needed about logs. The users, who are not necessarily familiar with the underlying details of the systems, query diagnostic information about the unexpected log lines while monitoring the health of systems. We have taken a complementary perspective and focused on information needs of an *engineer*. As opposed to users, engineers, responsible for maintaining the systems, not only need the diagnostic information (e.g., the context of error messages) but also execution details (e.g., interactions between components). Moreover, our study identified that performance data, which captures the duration of software and hardware actions, is essential for improvement and verification on timing performance for embedded systems.

We observe that developers often need to manually recover the links between different types of logs (see Section 2.2.4.1 and 2.2.5.3) to gain a more comprehensive understanding of an execution. Tool builders can consider recovering the links between different types of logs, e.g., using timestamps. Such tools would allow developers to inspect what functions and software actions are executed, and what critical functional data are produced when a specific high-level event occurs. In addition, we suggest tool builders to leverage semantic information (i.e., the textual elements in logs) to recover the links. Establishing links between software artifacts using the concept of semantic coupling (i.e., the semantic similarity between entities) has been demonstrated for many maintenance tasks such as traceability [33] and change impact analysis [186].

2.5.4.2 Log Comparison

Our study suggests that developers inspect not only one single log, but also a set of logs generated from multiple executions. To support developers in comparing logs, techniques have been developed to compare behavioral models extracted from logs generated from multiple executions [19, 43, 135, 254]. However, these techniques may not meet our developers' expectations because these tools require non-trivial configuration, e.g., the length of the minimal "interesting" sequence that differentiates two logs. For example, 2KDiff [19] compares two logs by highlighting the sequences of length k that belong to one log but not the other. All the differences based on the user-defined k are visualized on the models. Given the size of industrial logs (in gigabytes), inspecting such differences for two large executions might require significant cognitive effort to identify interesting information. Having concerns that it might require a lot of cognitive effort to identify interesting information from all the k-differences, Bao et al. [43] extend 2KDiff by taking the frequencies of behavior found in logs into account. The proposed tool visualizes statistically interesting differences by requesting users to set the target distance between probabilities,

and the statistical significance value, in addition to the parameter k. However, configuring such tools properly might be difficult and require iterations of parameter tuning because these parameters are related to the underlying statistical differencing model rather than to the nature of the software.

Based on the interviews, we believe that linking log differences to their sources and providing automatic categorization can help developers perform evolution tasks: whether a log difference is introduced by change of software code, logging code or variants of runtime behavior (e.g., concurrency). For example, when identifying a root cause of regression based on logs, developers can ignore the differences belonging to the categories of concurrency because these differences are not expected to influence the final outcome. To recognize the differences caused by code modifications such as refactoring and functional modifications, tool builders may consider to leverage existing tools from the fields of code differencing [116] and refactoring detection [371]. Chen et al. [76] demonstrate how to identify the change of logging code among all kind of code changes using regular expressions to match the source code. To identify log differences related to concurrency, tool builders can leverage previous work on log analysis that identifies interleaving events by logging the partial ordering relations between events [50, 106, 235]. The partial ordering relation between events can be captured with logical clock timestamps [113, 260] with which logical timestamps are generated for events in the system, and their causal relationship is determined by comparing those timestamps. Tool builders can consider adopting the methods from these studies to identify the differences caused by concurrency. The obtained information can be incorporated into log comparison to help developers recognize the useful log differences for their tasks.

2.5.4.3 Log Comprehension

Little research has investigated what hinder *developers* to comprehend logs. As discussed, Shang et al. [340] identified information needs of *system users* who monitor the health of systems and often need to query diagnostic information about unexpected log lines. Our study discusses the information needs by developers.

As discussed in Section 2.2.5.2, lack of familiarity with existing code and lack of domain knowledge can hinder log comprehension, especially for multidisciplinary systems: interpreting information from logs might require expertise from multiple engineering disciplines, while communicating with engineers of different disciplines is the commonly used method to obtain the expertise. Indeed, as observed (Section 2.2.5.2), working with logs from such systems requires software engineers to work with colleagues from other engineering disciplines to understand functional requirements of the systems and to interpret the information shown in logs. This observation is consistent with earlier findings [138]: the combination of software engineering with other engineering disciplines requires communication between engineers of different disciplines. In addition, we learned from the study at ASML (Section 2.2.5.2) and other ES companies (Section 2.3.2.2) that concurrent design and time-out mechanisms, implemented in embedded systems to optimize and limit software execution time [161, 345], also hinder log comprehension. We further

observed that interleaving of concurrent executions incurs challenges not only in program comprehension [28, 115] but also in log comprehension (see Section 2.2.5.2). Indeed, when inspecting logs, developers need to reconstruct the logical relations and order between interleaved function executions, as well as identify the differences between multiple executions that affect the execution outcome.

To cope with the complexity, we learned that experienced developers tend to adopt a top-down approach when inspecting logs. This concurs with a study on the relevance of application domain knowledge in program comprehension [339]—developers who are familiar with the application domain use a top-down approach to conserve efforts, developing a global hypothesis about the overall program based on high-level information, and then verify their hypotheses with more program details. The top-down method is known to be effective for system comprehension, which requires developers to understand the structure of the system: the main components and the communication paths between these components [221]. As opposed to code comprehension, system comprehension shifts the focus from the code to its structure, which is essential to comprehend large volumes of code. This is in line with our observation on developers' information needs—to understand the behavior of large scale software systems based on logs, developers need both structure information such as interactions between modules, and low level execution details (see Section 2.2.4).

Another coping strategy experienced developers adopt for log comprehension is to sketch and derive behavioral models based on logs. The derived models and patterns abstract the details of execution away, and are subsequently used for system comprehension, communications between team members and issue detection. There has been a lot of studies on automatically inferring models and patterns from logs [50, 52, 243, 259, 395, 406] for various software engineering activities. Beschastnikh et al. [50] designed a tool that helps developers comprehend distributed systems by visualizing the communication patterns between hosts. To help the debugging process, Mashhadi et al.[259] proposed a semi-automated technique that automatically abstracts the control flow from an execution trace with state machines, and then asks developers to interactively configure the tool to abstract the data-specific behavior. The empirical evidence collected from our study emphasizes the practical value of model inference techniques, and calls for more industrial applications of these existing techniques.

Our findings about log comprehension have two implications. First, our findings stress the importance of establishing multi-level abstraction of executions to support log inspection and log comparison. Many tools aim at abstracting away details from execution logs by deriving state machines [195, 240, 395], sets of temporal properties [219], and execution patterns [434]. These kinds of trace abstraction tools often rely on heuristics to create abstraction. For example, in order to extract a compact state machine model from traces, the underlying algorithms iteratively merge similar states based on heuristics, which can result in overgeneralization (e.g., containing behavior that is not observed in the trace) or under-generalization (e.g., without abstraction) in models [424]. Moreover, these tools provide only one level of abstraction, not

meeting the expectations of the interviewees (see Section 2.2.5.3) because the important information might be lost by showing only a certain level of detail. Several studies addressed this limitation [50, 112, 179] by allowing developers to inspect information at different levels of detail. However, these tools do not guide developers in information navigation, e.g., one needs to manually identify the relevant component interactions when analyzing issues with tools that generate sequence diagrams [50, 179].

This leads us to the second implication that tool builders may take domain knowledge into account, incorporating information from other sources (e.g., source code or bug reports) to guide developers to navigate through information at different abstractions for their tasks. In literature, the knowledge obtained from different software artifacts, e.g., source code and documentation, has been leveraged to assist software maintenance tasks, such as de-duplicating bug reports [11], ranking relevant files for bug reports [430], mining requirement knowledge [227] and code summarization [261].

We believe that a similar research effort is required to understand what code and domain knowledge developers need for log analysis and to leverage code and domain knowledge in log analysis tools. An example of such domain knowledge required for log analysis is the communication mechanism between components (see quote P15-4 in Section 2.2.5.2). We expect that augmenting logs with additional knowledge derived from source code and documentation, can reduce the time that developers spend in searching for information that is currently spread over multiple sources such as source code and documentation.

2.5.4.4 The Problem of Co-evolution

Our work extends the discussion of the evolution of logging code [76, 184, 432]. Kabinna [185] mined four open source projects and found that 20-45% of the logging statements are modified by developers at least once during their lifetime. Our study further provides empirical evidence on the co-evolution problems in software logging (Section 2.3.2.5), such as the challenges in maintaining the behavioral fingerprints of software issues derived from logs. Our finding implies the need for a deeper understanding of evolution of logging code and supporting the co-evolution of log-dependent entities. There have been some studies focusing on evolution of logging code. Studies of Yuan et al. [432] and Li et al. [225] have shown that most of the modifications of logging code are made to the content of log messages such as verbosity, variables and text. Li et al. [225] further discovered that logging code with similar context may need similar modifications. Therefore, the authors trained a machine learning model to predict modifications of logging code based on logging revisions, achieving a promising result. Unlike co-evolution of other software artifacts such as production and test code [435], metamodels and models [79, 265], and requirements of different components [108] that have been widely studied to help developer adapt these co-dependent artifacts, co-evolution of logging code and log-dependent artifacts is rarely addressed in the scientific literature. Research efforts are needed to aid developers in co-evolving log-dependent entities (e.g., behavioral fingerprints). Moreover, researchers should take the evolutionary nature of software logging into account when designing log

analysis techniques (e.g., to what extent would the accuracy of the machine learning models be affected by the evolution of logging code?).

2.5.4.5 Manual Analysis With Text Editors

Consistent with the previous study at Microsoft [44], we found that developers mainly use text editors for their log analysis activities. Given that many log analysis tools have been proposed, the observation implies a gap between research prototypes and industrial practice. The reason why developers use text-based tools could be that 1) practitioners are not aware of other existing techniques, 2) the existing techniques proposed by researchers cannot address the challenges that developers face, 3) the techniques that address the challenges have not been turned into products by tool builders, or 4) the tool adoption is hindered by extensive training and additional cost. To address these problems, we propose three types of studies for researchers to further identify and bridge the gaps between the state-of-practice and state-of-the-art.

First, we propose researchers to gain more insights into current practice of software logging. As collected by He et al. [154], over the years, commercial (e.g., Splunk [350]) and open-source tools (e.g., GrayLog [1]) have been made available for practitioners. Empirical studies should be conducted to get a comprehensive overview of the technical and non-technical influencing factors in the adoption of log analysis tools. One of the possible obstacles implied in our study is the difficulty of obtaining structured logs to enable the use of advanced analysis tools (i.e., the cost of parsing logs or migrating logging code for a large code base). The observation shows that many challenges stem from other steps of software logging (e.g., log instrumentation). Therefore, we believe research efforts should also be made to dive into the industrial practice of log instrumentation (e.g., logging approach and library) and management (e.g., log collection and analysis). For example, interesting research questions about log instrumentation could be: What logging methods and libraries do companies use? What is the rationale behind their decisions? What kind of challenges are they facing with the used methods? We believe that gaining more understanding about the practice, challenges and tool adoption is the first step toward solving the problems.

Second, we suggest researchers to create a mapping of the industrial challenges and the existing techniques that could potentially address the challenges. To achieve this goal, it is essential to obtain an overview of the state-of-the-art techniques that support log instrumentation and management through a literature study (e.g., systematic literature review and systematic mapping study). Many literature studies have been conducted to understand different activities in software logging and log analysis, such as log instrumentation [77] and log abstraction [107]. However, a systematic and in-depth mapping of current practice and existing techniques is still missing. We believe such mapping studies are important for researchers and tool builders to understand the gaps and the potential useful techniques that deserve further explorations and improvements.

However, the mapping studies may only give indications on promising techniques. In order to transfer the state-of-the-art techniques to practice,

it is important to conduct experimental studies in the field [359] where researchers can apply the techniques in a natural software development setting and study the possibility of integrating the techniques into the existing development process and infrastructure. To understand the impact of different solutions and environment settings, researchers can consider to conduct field experiments [359], which allows them to controls certain aspects of the setting (e.g., human factors). For example, to explore whether the state-of-the-art log comparison techniques can help developers efficiently identify the root cause of regressions, researchers can design a field experiment which involves the comparison of the promising techniques to the text-based comparison tools that developers used in their natural development setting. By experimenting with the techniques in the field, researchers can better understand the limitations of the techniques and the additional cost (e.g., training) the techniques may require. Iterations of refinement and experiment should be expected before the techniques are matured enough to be integrated into the development process.

With these three types of studies, we can better understand the nature of challenges in logging and log analysis, and the real-world design contexts, producing design knowledge to guide the development or improvement of techniques that address the identified challenges.

2.5.5 Technique Development at ASML

ASML has been developing techniques that address some of the challenges presented in our exploratory study at ASML (Section 2.2). Hooimeijer et al. [166] present a technique that infers multi-level state machine models from execution logs generated by component-based systems. Instead of using heuristics that often don't match system characteristics and are difficult to configure for practitioners, the technique learns multi-level state machine models that represent the behavior of systems, using the knowledge of the component-based software architecture. By showing the learned models to ASML developers, the authors validate that the models adequately provide ASML developers the software behavior abstraction that they currently lack (see discussion in Section 2.2.5.3).

As suggested by the interviewees in our study, such learned models can be used for software comprehension or serve as behavioral fingerprints of systems. To utilize the potential benefits of the learned multi-level state machine models, Hendriks et al. [159] extended this technique with a methodology that allows developers to automatically compare state machine models learned from execution logs, e.g., from different software versions, and to inspect the comparison results at various levels of details. By comparing software logs at six levels of abstraction with this methodology, developers can zoom in on relevant differences, and manage the complexity of large systems. The effectiveness of this methodology is demostrated with several case studies using ASML (sub-)systems. It is shown that the root cause of software regressions can be identified with the comparison methodology. Based on our study, we further suggest researchers to extend this methodology to categorize the behavioral differences obtained from the comparison, and to provide developers with actionable insights (as discussed in Section 2.5.4.2).

The fact that these techniques exist and have been validated through positive empirical evidence serves to confirm the value of our findings and implications in addressing log analysis challenges for embedded systems. Additionally, they serve as an illustration of how the results of our research can be applied to practical problem-solving.

2.6 Threats to Validity

As any empirical study, ours is subject to threats to validity.

Threats to **construct validity** examine the relation between the concept being studied and its observation. One threat could be that developers have different definitions of logs. To migrate this risk, we provided our definitions of software logs.

Threats to **internal validity** concern factors that might have influenced the results. First, developers might have misunderstood our interview questions. For the first study, we mitigate this risk by conducting a pilot interview with a developer who works at ASML, and rewording the questions as necessary. For the second study, we piloted both open and closed questions with an industrial embedded software developer and provided the explanation of individual options (i.e., codes) in the closed questions.

Second, our interviewees might hesitate to discuss the difficulties in their current practice or the issues in the tools they use. For example, it could be because that they were aware that the result will be published. We reduced their concern by explaining data privacy rights and guaranteeing them full anonymity. Third, the coding we applied to the interview transcripts is an interpretive procedure. Moreover, the coding tasks were single-handedly performed by the author of this thesis. This decision was made because of the technical knowledge, such as the state machine modeling language used by developers, required to interpret the information shared by our interviewees. To limit the researcher bias, we performed member checking. Developers were encouraged to correct our interpretations and add additional thoughts. In addition, the recent research at ASML related to log analysis techniques has shown the usefulness of our suggestions for researchers, increasing our confidence in our findings.

Threats to **external validity** concern the generalizability of our conclusions beyond the studied context. For our first study at ASML, we opted for convenience sampling, selecting the company that we have ongoing collaboration with. We expect that this company provides a representative context because the products of this company have been considered as a typical example of complex embedded systems in many studies [138]. In this study, we explored log analysis practices for control and metrology software, which is a typical module in complex embedded systems. To select interviewees from the division that is responsible for the module, we opted for purposive sampling [41] by encouraging each group lead from this division to recommend developers with different education backgrounds, genders, and roles. However, there is a risk that group leads might prioritize other factors (i.e., developers' availability) over diversity. To ensure saturation, we conducted interviews and coding tasks in an interleaved manner. We made a detailed report on the study

context to support the transfer of results to other similar contexts. To increase external validity of our findings, we conducted a dependent replication study at multiple companies using the same method (i.e., interviews). Convenience sampling is adopted to recruit companies that we have contact with. The selected companies from embedded domain are developing different kinds of embedded products. We discussed the contextual factors of logging in embedded systems (Section 2.4) that deserve further investigation to increase external validity and build theories. A future work could be conducting an independent replication study which uses different experimental procedures and involves more changing factors.

2.7 Conclusion

We explored how developers use logs in embedded software engineering by conducting an exploratory study at ASML. To refine the findings, the study was then replicated at four other companies. As the final result obtained by interviewing 39 developers in total, we identified four types of logs developers use, 21 purposes for which developers use logs, 14 types of information developers search in logs, 17 challenges faced by developers in log analysis, and six suggestions on tool support. The most prevalent information needs are related to *context of issues*, *state and interaction* and *timing performance*. We observed that text-based tools (e.g., Notepad++ and Linux `diff`) are commonly used by these embedded system companies for inspecting and comparing logs, despite that many academic and commercial log analysis tools have been proposed. Our study identifies that major challenges arise in log analysis due to poor log quality, insufficient expertise, and the high complexity of systems. Moreover, our study provides evidence that the evolution of logging code also introduces challenges. For example, log-dependent entities (e.g., log analysis tools) are affected by the change made to logging code.

Our study offers practical suggestions for logging practices, tool builders, and researchers. We recommend that practitioners design a systematic logging process that involves stakeholders, and that tool builders create advanced log comparison tools that categorize log differences to provide actionable insights for developers. Our study also highlights the need for further research in supporting the evolution of log-dependent entities.

3

Log Comparison: Understanding the State of the Art

Software logs capture the runtime behavior of systems and are analyzed by developers for various purposes, including root cause analysis and behavioral verification. Despite the availability of log comparison techniques, Chapter 2 discusses that developers often use text editors and face challenges. For instance, root cause analysis can be difficult due to an overwhelming number of differences resulting from concurrent executions, which may mask actual software bugs. To provide an overview of existing log comparison techniques and their limitations, we conduct a systematic literature review (the RQ2-3 highlighted in Figure 3.1 and present our findings in this chapter. Based on our analysis, we suggest further research to address the identified limitations.

3.1 Introduction

To facilitate various maintenance activities, software developers instrument software systems by adding logging statements in their software [44, 80, 224]. The generated software logs capture a rich amount of execution information of systems and are used for varied analyses, such as root cause analysis and liability analysis. In Chapter 2, we have identified that ASML developers compare logs generated from multiple versions of systems in their practices. The need for comparing logs is also identified in other big companies, e.g., Google [146] and Microsoft [44]. A typical purpose of log comparison is to analyze the root cause of software regressions. Developers generate execution logs from the passing and failing version of software. By comparing the

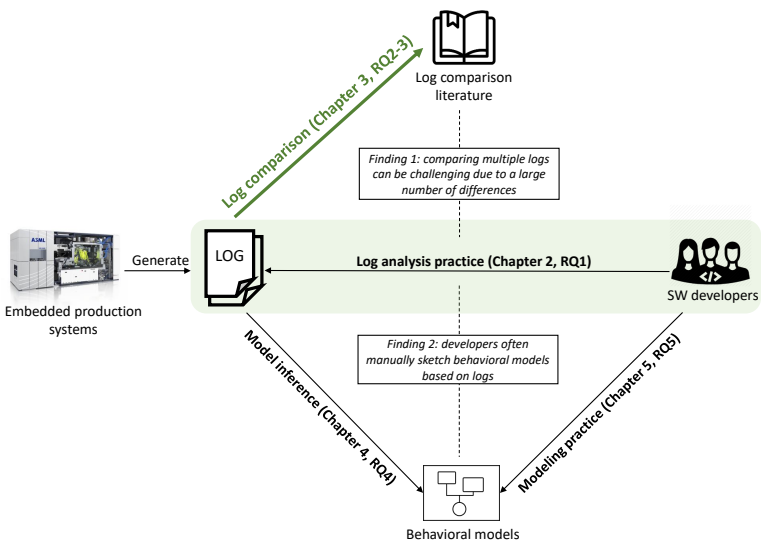


Figure 3.1: Research overview (RQ2-3)

generated logs, developers attempt to answer various questions that might help them identify the deviations and their causes, such as which events have executed in one execution but not the other, which sets of events have executed in different order in these two executions, and which set of events have consumed much more time in the failing execution.

According to our study presented in Chapter 2 and other empirical studies [146, 445] in literature, developers commonly use text-based comparison tools such as notepad++ [1] and Beyond Compare [2], while many log comparison techniques have been developed in literature over the years. These comparison tools treat two logs as textual files and compare them line by line without taking the semantics of logs, and the complex and evolving nature of software systems into account. Logs generated from two subsequent versions of systems could be different because of the changes made to the systems (e.g., refactoring and implementation of new features) or the natural differences of software executions (e.g., events occurring in different orders in different runs due to the concurrent nature of software). As reported in literature [146] and in Chapter 2, the overwhelming amount of irrelevant differences overshadows the difference that points to the actual software bugs. To tackle these problems, developers provide several suggestions for log comparison tools. For example, comparing logs and presenting log differences at different levels of abstraction is suggested as a way to deal with the large amount of log details. In order to help researchers improve log comparison techniques, we believe it is important to understand the existing techniques. Do the existing log comparison techniques target to help the software engineering (SE) activities where developers often need to compare logs? What are the characteristics of these techniques? Are these techniques

[1] `https://notepad-plus-plus.org/downloads/`. Accessed: September 2021

[2] `https://www.scootersoftware.com/` Accessed: September 2021

thoroughly evaluated to address real-world problems? Can these techniques address the challenges faced by developers in using text-based comparison tools? Understanding these questions can help researchers improve the existing techniques or propose new techniques to address the challenges reported by developers. To this end, we conducted a systematic literature review of log comparison techniques.

Our contributions are: (1) We provide an overview of log comparison techniques that are developed for different software engineering tasks; (2) We summarize these comparison techniques by analyzing the steps taken in comparison; (3) We report on the results of the technique evaluation, which shows the considered quality criteria in the evaluation and the maturity of these techniques; (4) We analyze whether these techniques can potentially address industrial challenges and meet developers' expectations.

The rest of the paper is organized as follows: Section 3.2 presents the research questions. In Section 3.3, we discuss the methodology we used in our research. Section 3.4 discusses the results and findings of our research. We then provide implications for researchers, tool builders, and practitioners in Section 3.5. In Section 3.6, we discuss related work. Threats to the validity of our research are addressed in Section 3.7. Finally, we conclude and summarize the results of our work in Section 3.8.

3.2 Research Questions

This literature study aims at providing an overview of state-of-the-art log comparison techniques and an in-depth analysis of these log comparison techniques for addressing industrial challenges. In this section, we discuss our research questions.

In this study, we consider logs as *textual files that record dynamic information produced by the execution of a(n instrumented) software system*. We do not provide a formal definition of a log, since papers vary greatly in how they describe the logs they consider. For example, Alcocer et al. [14] visualizes sequences of dynamic call graphs without elaborating on how these call graphs should be obtained, while in [442] the definition of a log remains implicit.

Comparing such logs is not only researched in software engineering, but also in various other areas of computer science, such as process management. For example, by comparing logs generated from medical information systems, a deviation in the executed process can be identified and further investigated. Although log comparison techniques from other research domains of computer science might not be directly applicable for software engineering tasks, an overview of the research fields for log comparison can help SE researchers apply log comparison techniques for supporting software engineering activities or apply their log comparison techniques in other problem domains. Therefore, we ask this question:

RQ2.1: *Which research fields in computer science are applying and developing log comparison tools?*

Next, we answer research questions to understand the development and application of techniques for SE activities. Developers may compare logs in

different problem contexts. As Gulzar et al. [146] indicated, Google developers compare logs after conducting differential testing for identifying the root cause of bugs. Our previous study at ASML (Chapter 2) found that developers compare logs not only for identifying the root cause of regression and flakiness, but also for verifying their software behavior with respect to their expectations. We aim to understand which SE activities the log comparison techniques from literature aim to help, assessing the alignment of practice and scientific literature. We study:

RQ2.2: *For which SE activities are log comparison techniques developed and applied?*

Different techniques may compare different kinds of information shown in logs and may use different methods to compare the same type of information. In the absence of an overview of methods used in these techniques, (1) practitioners and researchers have difficulty in selecting proper techniques for their problem scenarios, and (2) the existing methods might be reinvented while leaving key challenges less studied. We would like to study the methods of these log comparison techniques such as the form of inputs these techniques take, and the preprocessing steps these techniques require, and the log information these techniques compare. Therefore, we ask:

RQ2.3: *What methods do these log comparison techniques use?*

In order to transfer the state-of-the-art techniques to tools that can be used in the software engineering process, it is important for tool builders and practitioners to understand how and to what extent the techniques have been evaluated. We aim at gaining insights into the maturity of these techniques. Therefore, we ask:

RQ2.4: *How have log comparison techniques been evaluated?*

According to our study presented in Chapter 2 and other empirical studies [146, 445] in literature, the difficulty of log comparison stems from the complex nature of software systems and its evolution. To tackle the complexity, the developers involved in our previous study provide several suggestions for log comparison techniques (Chapter 2). First, these developers would like to inspect log differences with different levels of details. The multi-abstraction visualization can help developers comprehend complex information and guide them to inspect important low-level details. Second, concurrent executions can result in interleaving of events in logs. The differences caused by interleaving might be overwhelming and irrelevant for developers' analysis activities such as root cause analysis. Therefore, a way to recognize these irrelevant differences is required. Third, these developers expect advanced log comparison tools which leverage information from different sources (e.g., code) to help them interpret and comprehend the differences. Based on the challenges and suggestions derived from the empirical study, we ask:

RQ3: *How do log comparison techniques address industrial challenges identified in the literature?*

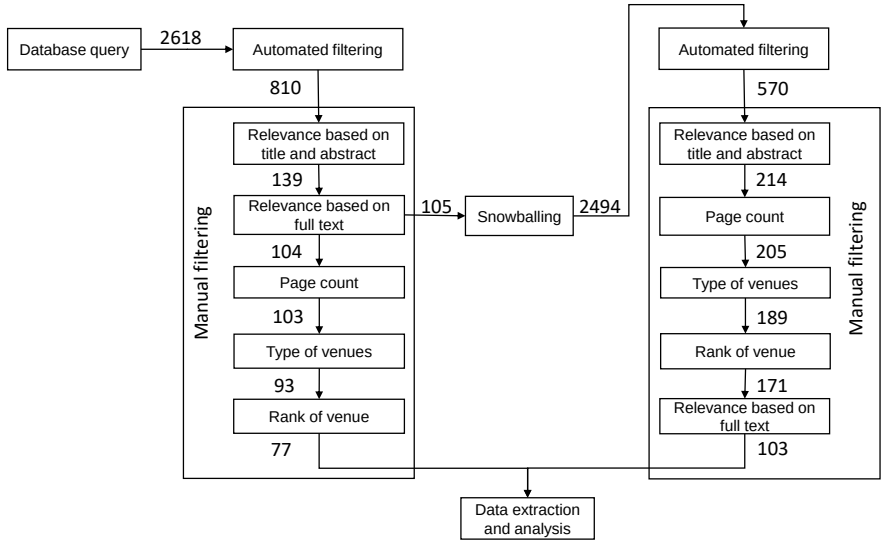


Figure 3.2: Process of literature study

- **RQ3.1:** *How do log comparison techniques enable multi-abstraction comparison?*
- **RQ3.2:** *How do log comparison techniques handle interleaving of events shown in logs?*
- **RQ3.3:** *How do log comparison techniques leverage additional information from other sources?*

By asking these question, we aim to gain a better understanding of to what extent the real-life challenges are addressed by state-of-the-art log comparison techniques. With this knowledge, we can then identify possible improvements for existing log comparison techniques.

3.3 Methodology

We follow the guidelines by Kitchenham and Charters [192] to perform the literature review. Figure 3.2 shows the process of this literature study. The process contains five high-level steps: database query, automated filtering, manual filtering, snowballing and analysis. Kitchenham and Charters focused on systematic literature review (SLR). In SLR, database search with keyword can lead to missing papers due to the choice of keywords and databases. In this study, we combine SLR and snowballing to compensate this shortcoming. In this section, we explain the methods used in these steps.

3.3.1 Database Query

To systematically collect relevant papers, we use the following digital libraries that are widely used in SE literature review: ACM Digital Library [2], Scopus [334], IEEE Xplore Digital Library [171], Springer Link Online Library [352], Wiley Online Library [412], and Elsevier ScienceDirect [333].

Table 3.1: Tokens in search queries

Token set	Tokens
Artifact (singular)	log, trace, execution
Artifact (plural)	logs, traces, executions
Verb (plural)	differ, compare, difference, differentiate
Verb (singular)	differs, compares, differences, differentiates
Deverbal noun (singular)	comparison, difference
Deverbal noun (plural)	comparisons, differences
Gerund	differencing, comparing

To develop the queries, we take the terms used in queries and the ways to compose queries into account. To familiarize with the terms used to describe the techniques, we first studied 10 papers about log comparison techniques that we collected and considered to be relevant studies. We learned that *trace* and *execution* are often used as synonyms of the *artifact log* and *comparison* and *differencing* are used interchangeably to refer to the *action* of looking for similarities and differences between the logs. Based on this observation, we formulate the queries for database search. Table 3.1 shows the tokens that represent *action* and *artifact*. We have seven token sets and each of the token set contains multiple tokens that represent the synonyms of a token. Furthermore, we also take the form of token (e.g., plural) into account.

There are many ways to compose a search query. Table 3.2 shows how the query is composed with these tokens in this study. As indicated by Landman et al. [202], forming a search query with operator *OR* may result in fewer results in some databases (e.g., IEEE Xplore). That is, query ‘‘`log comparison`’’ `OR` ‘‘`log differencing`’’ may return fewer papers than the results returned by separate queries ‘‘`log comparison`’’ and ‘‘`log differencing`’’. Therefore, we compiled multiple separate queries and each query is composed of tokens that represent the artifact (e.g., logs) and action (e.g., comparison). We then merged the search results to form the repository of papers. There are also multiple ways to compose the tokens that represent the artifact and action. The artifact and action can be glued with operator OR, e.g., ‘‘`log`’’ `OR` ‘‘`comparison`’’, which allows the other words to come between the artifact and action. However, since `log` is also a widely used mathematical term for representing logarithm, the search engines of these databases return a lot of irrelevant papers. For example, query ‘‘`log`’’ `OR` ‘‘`comparison`’’ results in more than 518K papers[3] in Springer Link Online Library and most of them are false positives. To minimize the noise, we decided to concatenate the tokens of artifact and action without operator *OR*. As shown in Table 3.2, a query can be constructed by concatenating a token from a set before operator * with a token from a set after operator *. We are aware that by concatenating tokens, we may miss the papers that state the artifact (e.g., *log*) and the action (e.g., *comparison*) separately (e.g., compare a set of logs). We mitigated this with a snowballing step explained later in Section 3.3.4. Following the query composition method, we obtained in total 78 queries.

[3] `https://link.springer.com/search?facet-discipline=%22Computer+Science%22&query=%22log%22+OR+%22comparison%22&flanguage=%22En%22`. Accessed: April 2021

Table 3.2: Composition of search queries. "*" represents concatenation. For two sets of token T1 and T2, the concatenation T1T2 consists of all tokens of the form vw where v is a token from T1 and w is a token from T2, or formally $T1 * T2 = \{vw : v \in T1, w \in T2\}$.

Concatenation	Example	#Queries
Artifact (singular) * Deverbal noun (singular)	log comparison	6
Artifact (singular)* Deverbal noun (plural)	log comparisons	6
Artifact (singular) * Gerund	log differencing	6
Gerund * Artifact (singular)	differencing log	6
Gerund * Artifact (plural)	comparing logs	6
Verb (plural) * Artifact (singular)	compare log	12
Verb (plural) * Artifact (plural)	compare logs	12
Verb (singular) * Artifact (singular)	compares log	12
Verb (singular) * Artifact (plural)	compares logs	12

Table 3.3: Results of database query

Database	#Papers	Used filter
ACM	258	Content type: Research article
IEEE	116	Publication type: Conferences and Journals
Scopus	854	Publication type: Conferences and Journals
ScienceDirect	378	Article type: Research article; Subject: Computer science
Springer	1286	Discipline: Computer science; Language: English
Wiley	119	Publication type: Journals; Subjects: Computer science
Total:		3011
Total (excluding duplications):		2618

We conducted the full-text query using the default search box on the databases. To reduce noise and exclude the papers not of interest, we applied filters when it is possible. We set publication type to conference and journal, research field to computer science, and language to English. Table 3.3 shows the query results. We identified the duplicated papers based on their DOI and titles. After removing duplicates, we obtained 2618 papers. We finished this round of paper collection in May 2021.

3.3.2 Automated Filtering

We manually inspected 50 papers from these 2618 papers and found that there are still many papers that are completely irrelevant to log comparison (e.g., papers from the field of Ultra-Wideband). To reduce the manual efforts required to select papers, we further applied an automated filtering approach proposed by Landman et al. [202].

The intuition behind this automated filtering approach is that if the paper is relevant to the studied topics, the related keywords should be frequently appearing in the introduction and conclusion of the paper. We implement this approach with five steps, as shown in Figure 3.3. First, we downloaded all the PDF files for the 2618 papers that we obtained from the database search. We extracted the text from the PDF files, and excluded the reference section. Next,

we extracted the first and last 15% of the text with the assumption that the introduction and conclusion of papers should be covered by the head and tail of the text. We then tokenized the extracted text and computed the frequency of each token. A paper is included for the subsequent manual filtering steps if its *head* and *tail* meet one of the criteria below:

- *Head* or *tail* contains at least one instance of the search query, i.e., $\{q \in Q | head.count(q) > 0 \vee tail.count(q) > 0\} \neq \emptyset$ where Q is the set of search queries that we formulated based on Table 3.2.
- Both action and artifact tokens appear more than once in *head* and *tail*, i.e., $\{at \in AT \wedge ac \in AC | head.count(at) > 1 \wedge head.count(ac) > 1 \wedge tail.count(at) > 1 \wedge tail.count(ac) > 1\} \neq \emptyset$ where AT and AC are the sets of artifact and action tokens, respectively.

After filtering based on the criteria above, we obtained 810 papers. To check whether the approach is reliable, we randomly sampled 50 papers that are excluded by the filter. The author of this thesis checked the abstracts of these papers. It turned out that these 50 papers are indeed irrelevant to log comparison techniques, which increases our confidence on the applied approach.

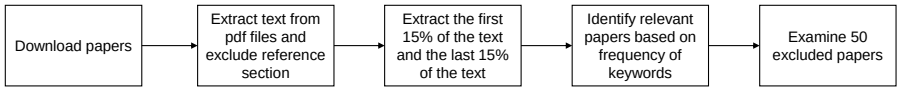


Figure 3.3: Steps in automated filtering

3.3.3 Manual Filtering

After automated filtering, we started with the manual filtering process, which requires us to manually collect information from papers to decide the relevance of papers. Three raters are involved in this manual process. We defined a set of inclusion and exclusion criteria for this manual filtering process (Table 3.4). Note that we include only offline log comparison techniques (IC4 in Table 3.4) because the challenges under study (RQ3) are identified in industrial settings where developers usually collect logs from executions and then compare [424].

As shown in the Figure 3.2, the manual filtering process contains steps that check the relevance of papers (i.e., relevance of papers based on title, abstract and full text) and quality of papers (i.e., page count, types of venues and rank of venues). Next, we explain these steps.

Relevance of papers We filtered on the relevance of papers to log comparison. First, we selected the relevant papers based on their title and abstract. We adopted a web application developed by Lin et al. [230] to facilitate the manual filtering process. Using this app, we assigned a batch of papers to three raters. For each paper, the raters need to select from options "included", "discarded" or "secondary study". The secondary study is the meta-analysis of the studied topic, e.g., literature review, which may refer to relevant papers. The obtained secondary studies have been used in the snowballing process to identify additional papers. We included the papers for the next round if

the information shown in abstract and title is not sufficient to determine its relevance. To reduce the chance that a paper was discarded by mistake, each paper has been labeled by at least two raters. To develop agreement on the relevance of papers, each of the raters labeled a same batch of 30 papers individually. The raters then resolved the disagreements together. With the discussion, the common understanding of the relevance of papers and the selection criteria was established. We repeated this process three times and labeled in total 90 papers together with the three raters. In the last iteration, we reached a Fleiss Kappa value of 0.83 which is considered to be almost perfect by the commonly used standard [262]. We then continued with the rest of the papers. Each paper has been labeled by two raters. When a conflict occurs, the third rater is involved for resolution. We further determined the relevance of papers based on the information shown in the full text of papers. Similarly to the previous step, each paper has been considered by two raters, and the conflicts are resolved by the third rater.

Quality of papers We determined the quality of papers based on three properties: page count, type of venue and rank of venue. To include only the full research publication for analysis, we exclude the papers that have fewer than 4 pages [392]. Further, we manually examine the types of venue and included only the papers published at academic conferences or journal. To further control the quality of the selected papers, we filtered papers based on the rank of the venues. To mitigate the bias of a single venue ranking system, we extracted the ranks of venues from multiple ranking systems: Conference Ranks [4] and GGS [5] GGS collects ranks from The CORE 2021 Conference Rating, Microsoft Academic, and LiveSHINE. After collecting the rank information from different sources, we post-processed the information based on several rules. First, we normalized the ranking into classes A, B and C. That is, a conference is considered to be A rank if the rank for a conference is A^+,A^{++},A or A^- in these ranking systems. Second, we labeled a venue as "unranked" if we cannot find its rank in these systems. Third, a conference or journal might be ranked differently in different systems. We adopted the majority rule to decide the final rank of the venue. For example, conference *Fundamental Approaches to Software Engineering* is ranked as class B in CORE, and as A in both LiveSHINE and Microsoft Academic. Therefore, the final rank of this conference is A. Finally, we included all the papers that are ranked A, B or unranked. We include unranked venues because a venue could still be primary even though it is not ranked in these systems.

As shown in Figure 3.2 after relevance assessment, exclusion of short papers, and the venue-based filtering we kept 77 papers.

3.3.4 Snowballing

To mitigate the risk of missing relevant papers, we conducted snowballing to get papers that are cited by the papers in our paper repository (i.e., backward snowballing) and the papers that cite the papers in our paper repository (i.e., forward snowballing). As can be seen in Figure 3.2, we took 105 papers that we

[4] `http://www.conferenceranks.com/`. Accessed: July 2021

[5] `https://scie.lcc.uma.es:8443/gii-grin-scie-rating/`. Accessed: July 2021

Table 3.4: Inclusion and exclusion criteria

	Inclusion	**Rationale**
IC1	The paper must be Peer-reviewed and published at conferences, or journals.	Quality
IC2	The traces/logs should be generated from instrumented software systems and capture the execution of software systems.	Scope of study
IC3	The study must adopt or develop at least one log comparison technique.	
IC4	The adopted or developed technique must be an offline technique, i.e., logs are collected from executions and compared offline for analysis.	
IC5	The adopted or developed techniques are used for computer science applications.	
	Exclusion	**Rationale**
IE1	The paper is written in a language other than English.	Generalizability
IE2	The paper has been extended to a journal article.	Redundancy
IE3	The paper is not a full research publication (e.g., abstract, doctoral symposium articles, presentations, posters, book, chapter, technical report and white paper etc.)	Quality
IE4	The study does not describe the approach unless it is a secondary study.	

obtained after filtering based on full text as the seed papers for snowballing, rather than the 77 papers we finally obtained with all filtering steps. The rationale is that, although the excluded papers do not meet our criteria in terms of quality, they may still cite or be cited by some relevant quality papers.

We used Semantic Scholar [336] to facilitate the process of snowballing. Semantic Scholar is an artificial-intelligence backed search engine for academic papers, that has been empirically shown to be suitable for literature studies in SE [150]. Hannousse [150] conducted a coverage test using 20 SE systematic literature studies and found that Semantic Scholar covers 98.88% of the papers included in these literature studies. With Semantic Scholar, the author can replicate 13 studies fully and more than 90% for the 7 remaining studies. Moreover, it leverages artificial intelligence techniques to provide more information about the relations between a paper and its cited and citing papers. For each seed paper, Semantic Scholar provides meta-data,[6] including a list of references (i.e., cited papers) and citations (i.e., citing papers). Each reference and citation is further labeled as *background*, *method*, or/and *results*, indicating its relation with the seed paper.[7] For example, a cited paper with label *method* is cited by the seed paper to describe methods, while a citing paper with label *method* means it cites the seed paper to describe methods. Furthermore, based on features such as where the citation appears in the body of the paper, Semantic Scholar labeled a reference or citation as a *highly influential paper* if it highly influences the seed paper or is highly influenced by the seed paper [385]. In our study, for each seed paper, we included the citing and cited papers that are

[6] `https://api.semanticscholar.org/graph/v1#tag/paper`. Accessed: July 2021

[7] `https://www.semanticscholar.org/product/tutorials`. Accessed: July 2021

Table 3.5: Result of snowballing

#seed papers	105
#cited papers	4175
#citing papers	2014
#unique papers	4267
#selected papers	2494

labeled with *method*, *results* or *highly influential paper*. We excluded the papers labeled with *background* because such papers only provide the background information of the topic or study, and are less likely to present log comparison techniques. Furthermore, we further excluded papers from the fields other than computer science based on the field classification provided by Semantic Scholar.

We used the APIs [8] provided by Semantic Scholar to query the data. Table 3.5 shows the number of papers we obtained from snowballing. After merging citing and cited papers, and removing duplications, we obtain additional 4267 papers from backward and forward snowballing. By applying the criteria based on the filters provided by Semantic Scholar, we ended up with 2494 papers. We then followed the automated and manual filtering process (See Section 3.3.2 and 3.3.3) to select relevant papers from this set of snowballing papers. Figure 3.2 shows the number of papers we included in each step. As surveyed by Wohlin et al. [415], the most common way of paper search in software engineering is the database search followed by the manual search. Snowballing is less adopted according to Wohlin et al., however, a single iteration of snowballing is shown to be effective. Since we do not plan on another iteration of snowballing, we applied the quality filter (i.e., page count, type of venue and rank of venue) before the relevance filter based on full text to reduce the manual efforts in inspecting the full-text of papers.

3.3.5 Data Extraction and Analysis

With the methods of database query, filtering and snowballing described above, we obtained 180 papers in total; 77 papers from the first round and additional 103 papers from snowballing. To assess the quality of our dataset, we examined whether the 10 papers that we studied for familiarizing the terms used in log comparison papers are found by our paper search. The assessment shows that all these 10 papers are included in our dataset.

We compiled a list of sub-questions with which we analyze each paper to answer our research questions. Table 3.6 lists these sub-questions and the corresponding research questions.

3.3.5.1 Field Classification (RQ2.1)

We classified the field of log comparison techniques based on ACM Computing Classification System[9] (CCS) which is a subject classification system for the computing field. It is a tree with 13 first-level nodes representing high-level fields, and each node has multiple child nodes representing subfields. For

[8] `https://www.semanticscholar.org/product/api`. Accessed: July 2021

[9] `https://dl.acm.org/ccs`. Accessed: September 2021

Table 3.6: Sub-questions for paper analysis

Research field (RQ2.1)
1. For which research field are log comparison techniques applied and developed?
SE activities (RQ2.2)
2. What SE activities does the proposed technique intend to support?
Methods (RQ2.3)
3. What are the inputs of the comparison technique?
4. What log preprocessing steps does the technique perform?
5. What log information is represented in the step of log representation?
6. What intermediate log representations does the technique use?
7. What post-processing steps does the technique perform?
Evaluation approach (RQ2.4)
8. What research methods are used in evaluation?
9. Which groups of practitioner participates in the evaluation if human is involved in evaluation?
Industrial challenges (RQ3)
10. How does the technique provide multi-abstraction comparison?
11. How does the technique deal with the interleaving caused by non-determinism?
12. What additional information is used to aid in log comaprison?
13. How does the technique use the additional information?

example, field *Software and its engineering* is a six-level tree. We would like to classify the papers into the 13 first-level nodes. We first examined all the papers to check for CCS concepts added by the authors of the papers. We extracted CCS concepts if they were available for a paper. A paper could be assigned multiple CCS concepts. In such cases, we considered the first CCS concept as the most relevant concept, as authors of ACM publications are suggested by the guideline about the use of ACM CCS [10] to rank CCS concepts based on their relevance. For the papers that do not contain CCS concepts, we labeled them manually. First, two raters familiarized themselves with the classification tree. Second, for each paper, these two raters independently read the title and abstract, and then selected a first-level node of the classification scheme which is most relevant to the paper by looking at the lower levels under each first-level node. Any disagreements were resolved by a discussion between the two raters who were involved in this classification task.

3.3.5.2 SE Activities (RQ2.2)

In our previous study of log analysis practice in industry [427], we derived a set of SE activities for which developers analyze logs. We use this set of SE activities as the initial classification schema and extend it when a new activity appears in the collected papers. The author of this thesis analyzed the papers to classify the collected papers based on the SE activities that the developed techniques attempt to help.

3.3.5.3 Methods (RQ2.3)

We studied the inputs of techniques and workflow of log comparison.

[10] `https://www.acm.org/binaries/content/assets/publications/article-templates/ccs-howto-v6-12jan2015.docx`. Accessed: September 2021

Inputs of Techniques There is a considerable overlap between the fields of log comparison and model comparison. The papers in the intersection of these two fields present techniques that consider models as a representation of logs: while models are typically stored in a different format, and are often visual, they still encode the software behavior observed in logs. That is, for this subset of log comparison techniques, their inputs could be models derived from logs. We classified the inputs of these techniques into the following groups:

- *Log-log:* the techniques that belong to this category perform comparison at the level of logs. Therefore, the inputs of techniques are multiple logs. The logs could be stored as a raw textual file or with a certain structure in a database.
- *Model-model:* the techniques that belong to this category assume that models derived from logs are given as inputs. The comparison is performed at the level of models.
- *Log-model:* the techniques that belong to this category compare logs against models that were derived from logs.

Workflow of log comparison We focus on several important steps in log comparison. A typical workflow is presented in Figure 3.4.

The typical steps in this workflow are:

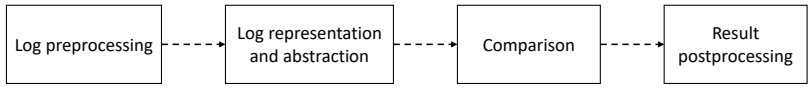


Figure 3.4: Workflow of log comparison. Note that log pre-processing, log representation and abstraction, and result post-processing might not be described in the studied papers.

- *Log pre-processing.* The amount of information to be compared is reduced in this step, easing the comparison step. In the pre-processing step, the textual form of logs is not changed.
- *Log representation and abstraction.* To further ease comparison by providing a formal structure for log information, logs might be abstracted. This step abstracts log information with models (i.e., a *different form of representation than text*).
- *Comparison.* In this step, logs are compared, either in the form of text or other representations obtained from the previous step.
- *Result post-processing.* The result of log comparison could contain an overwhelming amount of differences. To reduce, prioritize or aggregate information, a post-processing step is sometimes performed.

Note that log pre-processing, log representation and abstraction and result post-processing are not necessary steps and might not be described in the studied papers. The techniques that take model-model as input assume models are given. In this case, log pre-processing, and log representation and abstraction is not required. Furthermore, for the techniques that take log-log or log-model as input, the log pre-processing step might be omitted. The author of this thesis

analyzed the papers to check if these steps are explained and how the steps are taken to answer questions *4-7* (shown in Table 3.6).

3.3.5.4 Evaluation (RQ2.4)

We employed a research strategy framework described by Storey et al. [359]. The research strategy framework provides a classification schema for empirical strategies:

- *Data strategies.* Data strategies refer to evaluation methods that rely primarily on generated or simulated data.
- *Lab strategies.* Lab strategies typically involve hypothesis testing with controlled experiments in lab or experimental simulation that rely on an environment that mimics a real-life environment. Human participants are often involved in these experiments.
- *Field strategies.* Field strategies involve technique evaluation in a natural software development setting. In these strategies, researchers may observe developers using the techniques without any explicit interventions or set up controlled experiments that require developers to use the techniques solving real-world problems.
- *Respondent strategies.* Respondent strategies are used to gather insights from developers about the techniques under evaluation. The strategies may employ survey and interviews to gather opinions from developers about the effectiveness of the log comparison technique.

We chose this framework for categorizing evaluation methods (for Question *8* shown in Table 3.6) because each empirical strategy has strengths and weaknesses to consider in terms of research quality criteria [359]. Field strategies provide a higher realism while sacrificing generalizability. Data strategies have a strength of precision and generalizability, but a weakness of low control over human factors. Lab strategies allow researchers to control influencing factors at the price of realism and generalizability. Respondent strategies may increase generalizability (e.g., if a wide sample is recruited) at the expense of lower realism. Therefore, a mixed method (e.g., using field and data strategies to achieve both high realism and generalizability) is recommended [359]. By categorizing evaluation methods based on this framework, we aim to provide insights into the quality criteria considered in the evaluation of log comparison techniques. Next, we answer question *9* (shown in Table 3.6) to understand which groups of participants are involved in the evaluation. Answering this question further provides some insights into realism and generalizability of the technique evaluation. The analysis for these questions is conducted by the author of this thesis.

3.3.5.5 Industrial Challenges (RQ3)

As we discussed in Section 3.2, comparing logs at multiple levels of abstraction is favored. We answer Question *10* (shown in Table 3.6) by studying whether the technique compares and presents log information at different levels of abstraction, while preserving the link between information shown in different abstraction layers so that developers can drill down to the log details. Non-determinism results in interleaving of logging events, therefore, it is one of the main sources of irrelevant differences when comparing logs for troubleshooting functional issues [146, 427]. We study whether and how these log comparison

techniques take non-determinism into account (Question *11*). Another industrial challenge in log comparison is leveraging additional information to aid in log comparison. The information shown in logs is limited to the runtime behavior of systems. Often, inspecting log differences does not sufficiently help developers comprehend the changes of software systems. We, therefore, identify what types of additional information are used and how they are used in these techniques (Question *12* and *13*). The author of this thesis analyzed the discussion related to these questions. Note that we only examine the papers that explicitly discuss these topics. It could be that the proposed technique can potentially handle event interleaving, while not explicitly mentioned in the paper.

3.4 Results

In this section, we present the answers to the research questions.

3.4.1 Research Fields (RQ2.1)

Table 3.7 shows the classification obtained for RQ2.1. In total, 180 relevant papers are classified into 6 categories from ACM CCS. Nearly half of the papers (45%) are from the field of **Software and its engineering**. In this field, log comparison techniques have been mostly developed to support software engineering activities such as testing or issue analysis. In Section 3.4.2, we present our answer to RQ2.2 for which we detail the software engineering activities these log comparison techniques are developed for.

Table 3.7: Field classification

Fields (#papers)	References
Software and its engineering (81)	[14, 15, 16, 17, 19, 27, 36, 42, 43, 47, 50, 73, 78, 89, 94, 97, 98, 104, 106, 121, 135, 141, 142, 143, 145, 147, 151, 152, 155, 163, 172, 173, 200, 209, 231, 232, 242, 245, 253, 256, 263, 266, 268, 269, 270, 271, 276, 283, 284, 294, 302, 307, 310, 314, 315, 326, 327, 328, 338, 351, 362, 363, 367, 369, 375, 396, 398, 399, 400, 402, 403, 417, 420, 440, 441, 444, 445, 449]
Information systems (63)	[3, 7, 8, 18, 34, 40, 51, 55, 56, 57, 58, 59, 63, 66, 67, 74, 91, 92, 93, 99, 100, 102, 110, 118, 130, 131, 139, 165, 211, 213, 214, 215, 216, 222, 233, 234, 236, 237, 248, 249, 277, 278, 281, 299, 303, 304, 317, 337, 349, 366, 373, 374, 381, 384, 387, 389, 397, 404, 405, 416, 423, 437, 443]
Applied computing (16)	[6, 9, 13, 61, 86, 198, 220, 251, 257, 286, 288, 289, 301, 316, 364, 368]
Security and privacy (14)	[10, 26, 35, 120, 126, 149, 181, 191, 226, 252, 267, 279, 422, 447]
Human-centered computing (3)	[160, 244, 305]
Networks (3)	[194, 290, 341]

Around 35% of papers are from the field of **Information systems**. Log comparison techniques have been used for studies of database or process

management. Typically, such studies propose techniques to compare process models generated from logs collected from the execution of information systems such as banking systems. Such logs often capture how users interact with an information system. The papers from this field tend to focus on presenting generic techniques for processing such logs or by a demonstration of the feasibility with a certain application. An example is the work by De Leoni et al. [91] where an alignment-based approach for log comparison is proposed, and its feasibility is demonstrated with an example. In this example, logs are electronic patient records that describe processes of hospital organizations. The comparison of logs helps improve and optimize the hospital system. Different from log comparison techniques in the field of Software and its engineering, techniques from Information systems compare logs to analyze the differences of business and organizational processes rather than the behavioral differences of software systems.

The rest of papers (20%) are from the fields of **Applied computing**, **Security and privacy**, **Human-centered computing** and **Networks**. In **Applied computing**, log comparison techniques are applied to identify the variants of workflows or processes of information systems. Different from log comparison techniques in the field of **Information systems**, studies belonging to the **Applied computing** field focus on solving problems with existing process comparison techniques rather than proposing new techniques. For example, Martínez-Carrascal et al. [257] applied process mining techniques to analyze differences between passing and failing students in a blended-learning course. In the field of **Security and privacy** log comparison is used for malware analysis. Typically, the stakeholder of techniques is a security analyst. For example, Li et al. [226] built intrusion detection systems (IDSs). IDSs compare traces with models derived from event logs that are generated during the applications' normal operation and detect traces of malicious activities targeted against the network and its resources. Log comparison techniques have also been further developed in the field of **Human-centered computing** where visualization of log differences is the focus. For example, Low et al. [244] propose a visualization technique to provide targeted analysis of resource reallocation and activity rescheduling. With the proposed visualizations, resource- and time-related changes can be identified, and subsequently actionable items can be derived for business process management in practice. Finally, three papers are related to log comparison for **Network simulation**. For example, Kremer et al. [194] leverage root cause analysis technique from software engineering for comparing traces issued from different simulations and real experiments.

RQ2.1 summary:

The majority of log comparison techniques are developed in the research field of software engineering and information systems.

3.4.2 Software Engineering Activities (RQ2.2)

Log comparison is used for multiple software engineering activities. Table 3.8 shows the SE activities derived from the 81 SE related papers. Note that a

Table 3.8: Software engineering activities that log differencing techniques address

SE activities	definition	#papers
Issue analysis	In this category, techniques aim to help developers perform activities related to analyzing software issues (e.g., fault localization and debugging performance issues).	48
Verification and improvement	In this category, techniques aim to help developers verify and improve software systems for better quality (e.g., software modernization).	20
Comprehension	In this category, techniques aim to help developers comprehend the behavior of software systems.	13
Testing	In this category, techniques aim to help developers perform activities related to software testing (e.g., improvement of test code).	9
Research	In this category, techniques are used by researchers in software engineering research.	3
Deployment	In this category, techniques aim to help developers deploy software on a target device.	2
Program repair	In this category, the technique is used to determine the correctness of automatic program repair.	1

paper might be related to more than one SE activity because the presented log comparison technique might be applicable for multiple SE activities. It can be seen that more than half of papers (n=48) have considered issue analysis as the application for the developed log comparison techniques. Issue analysis is an SE activity that involves several sub-activities, such as fault localization [98], error propagation analysis [327] and debugging [145]. Most papers position their techniques in a general context of analyzing software issues, while only a small share of papers explicitly discuss the type of applications and software issues the log differencing techniques aim to address. Six studies [50, 121, 209, 235, 270, 328] develop techniques for analyzing issues in distributed systems. Two studies [403, 440] specifically investigate concurrency bugs (e.g., race condition and flaky bugs). Seven studies [14, 47, 104, 283, 302, 328, 402] focus on log differencing for analyzing performance issues.

There are 20 papers presenting log differencing techniques for verification and improvement activities, such as software modernization [94], anomaly detection [36, 42, 121, 135, 152, 189, 256, 263, 268, 269], and performance evaluation [47, 245, 276, 351]. 13 papers present techniques for software comprehension activities such as localizing features [106, 266] and understanding evolution of software [43, 172, 253, 396]. There are nine papers presenting log differencing techniques for testing activities, such as improvement of test code [268, 398] and prioritization of test cases [269]. Interestingly, log differencing techniques are used not only for SE activities but also SE research. Ardimento et al. [27] use logs to record how developers perform coding activities and compare logs to identify behavioral similarities and differences between developers. Pradel et al. [307] propose an approach that evaluate the state-of-the-art specification mining techniques. The approach compares models learned from logs with reference models to examine the

accuracy of the learned models by these specification mining techniques. Similarly, Walkinshaw et al. [396] develop a model-based comparison approach which can be used not only for comprehending the evolution of software systems but also for evaluating the accuracy of specification mining techniques. Furthermore, two papers motivate the use case of their techniques with deployment. As an example described in [375], when tuning a deployment of an email server, it is useful for engineers to identify the previous deployments with similar operational profile for solving workload problems (e.g., slow response time under a particular workload) encountered in the deployment activity. Lastly, log comparison has also been used to select patches generated by automatic program repair techniques. By comparing the logs obtained before and after the patch for each test, Xiong et al. [420] determine the correctness of the generated patches.

RQ2.2 summary:

Issue analysis is considered as the main use case for log differencing in the literature. Interestingly, log differencing techniques have been applied for SE research to study how junior and senior developers differ in terms of their coding behavior.

3.4.3 Log Comparison Methods (RQ2.3)

Next, we discuss our analysis of log comparison methods. We start with the inputs of these comparison techniques and then continue with log pre-processing, compared log information, log comparison, and result post-processing.

3.4.3.1 Inputs of Log Comparison

The result shows that majority of the papers (n=68) present techniques belonging to category *log-log*. A small share of papers present techniques that take *log-model* (n=8) and *model-model* (n=5) as inputs. Among the *log-log* papers, four papers [141, 145, 269, 375] present techniques that compare a single log against a set of logs and output a single log or a subset of logs as the comparison result. These papers describe techniques that share a practical purpose—identifying reoccurring patterns with historical dataset. For example, Gu et al. [145] use a bug trace to exhaustively search its database for similar bugs and return their bug reports by comparing the bug trace with a set of traces that are associated to known bugs.

3.4.3.2 Log Pre-processing

To reduce the amount of information being compared, textual logs should be pre-processed. Through reviewing the literature, we identified three categories of preprocessing activities, namely, parsing, segmentation, and noise reduction.

Parsing. Parsing is the process of structuring log data into chunks of information that are easier to manipulate. Typically, this process involves the identification of the static and dynamic components in log statements. The

dynamic information, such as usernames, IP addresses and job IDs, may vary for each occurrence of a particular execution event. To reduce the noise in comparison, it is essential to remove the dynamic information considered irrelevant. The traditional way of parsing is often done with handcrafted regular expression [42, 121, 135, 256, 284], requiring domain knowledge and constant update of regular expressions. To reduce the manual efforts of creating regular expressions that specify how to separate the constant part, several studies apply automatic parsing techniques. Syer et al. [367] and Thakkar et al. [375] use a code clone technique to identify for each log line the variation points relative to their log lines. Fu et al. [121] apply a log clustering technique to group log messages printed by the same statement together, and then find their common part as the static information. Tak et al. [369] apply two methods, source code analysis and log clustering techniques, to discover log templates. Identifying log templates is challenging because of the dynamic nature of the system development [135] where logging format might not be standardized. Moreover, modern systems are heterogeneous, consisting of many components implemented with different program languages. The format of log statements in different components can be different (see discussion in Section 2.2.5.2).

Segmentation. A log file collected from the field often contains day-long executions. To tackle the problem, a log file should be segmented into multiple execution logs, and each of them represents the execution of individual system tasks. The segmented log is the basic unit for the subsequent activities, such as log abstraction or comparison. To distinguish multiple executions in a long trace, Doray and Dagenais [104] employ Linux kernel events *syscall_exit_accept* (generated when a connection is accepted on a socket) and *syscall_entry_shutdown* (generated when a connection is closed) as markers of the start and end of executions. For the same purpose, Bao et al. [42] extract traces from a log according to the reachability relations revealed in reachability graph obtained from code analysis. Often, a single execution may contain several phases and repeated behavior patterns. Wang et al. [400] and Mohror et al. [276] use control structure boundaries such as procedure calls and loop boundaries as markers to identify the phases and repeated patterns. Modern software systems often consist of multiple entities (e.g., components, threads, tasks or user actions) that produce interleaving events, and different entities often aggregate their execution logs into a single log file. To segment a log into chunks that represent the behavior of single entities, the logged identifiers [121, 163, 256] are used. For example, Fu et al. [121] obtain logs from Hadoop systems where log messages for different tasks are interleaved. By segmenting based on the task IDs, sequential log message sequences are obtained. However, as discussed by Mariani et al. [256], the comparison of the logs divided by identifiers is effective in identifying differences related to single entities (e.g., single components or single user actions), but is not effective in identifying important differences related to the interaction and synchronization between multiple entities (e.g., interleaving of events from multiple components). Therefore, dedicated methods have been developed to deal with event interleaving. We elaborate on them in Section 3.4.5.

Noise reduction Some events are considered irrelevant for behavior comparison. To further reduce noise, these events are removed from logs [17, 47, 141, 172, 444]. For example, Idris et al. [172] and Alimadadi et al. [17] remove library method calls based on naming conventions. Hashing has also been considered as a method to reduce the volume of information. For example, Ramanathan et al. [314] use hashes to reduce the amount of space required to store the trace being proportional to the number of instructions executed rather than being proportional to the number of lines in the program.

3.4.3.3 Compared Log Information

We identify five categories of log information that are being compared by these techniques. Figure 3.5 shows the number of papers presenting techniques that compare each type of information. Note that a technique may compare more than one type of information. It can be seen that the majority of techniques compare ordering information such as the order of events, methods, or executed statements shown in logs. The ordering information reveals the execution flows, and its differences between executions reflect different execution paths that a software system takes at runtime. The second most frequently compared information is occurrence and frequency. These techniques consider log as a sequence of events, where the occurrence and frequency of each log event, words in log events, or a sequence of events is counted (i.e., with n-gram where ordering information is considered up to the length of n). The distributions of events in logs are then compared to identify deviations. For example, in the work by Zhou et al. [444], logs are treated as an unordered collection of individual lines (where the ordering information is ignored) for comparison. In this work, log lines are considered similar if they share words that are uncommon in the collection or corpus of all lines. Resource consumption and value information is compared by techniques presented in 18 papers. Particularly, resource consumption information (e.g., execution time of methods and CPU consumption) reveals the performance aspect of systems, therefore, is often used to identify performance deviations and regression of two executions. Value information includes the value of variables, return value of methods, and evaluation of branches. By comparing the value information obtained from two executions, differences that lie in the execution results, decisions and states of a certain execution point can be identified. For example, in the work by Sumner et al. [363], the value of a set of pre-defined variables is recorded to examine program states at each execution point of two executions. Less often, one compares data dependency information, which includes the data dependency between execution statements. For example, Tonella et al. [403] compare data dependency of passing and failing executions to produce a causal execution path that leads from the program point representing the root cause of failure to the program point at which the failure is detected.

3.4.3.4 Log Comparison

Table 3.9 shows the identified categories and the papers that belong to these categories. Note that a paper can present a hybrid technique based on multiple approaches, indicating the orthogonal nature of these approaches. Figure 3.6 shows the number of papers for each approach and hybrid solutions. From Table 3.9 and Figure 3.6, it can be seen that nearly half of the papers present

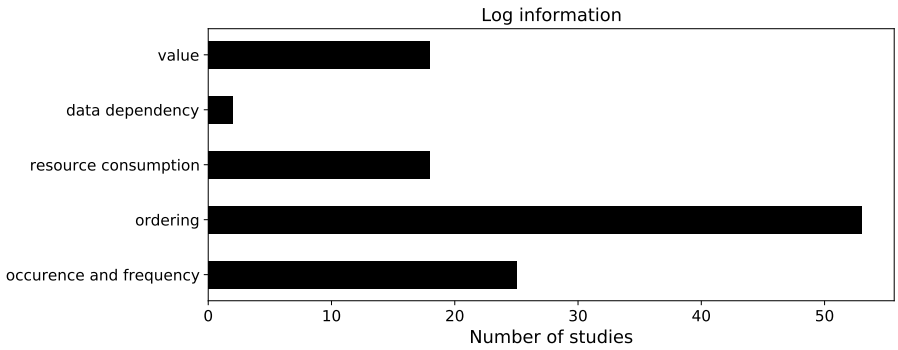


Figure 3.5: Number of papers presenting techniques that compare different kinds of log information

techniques that are based on *Alignment and matching* (n=40) or *Metric* (n=34). A significant number of papers present techniques that are based on *Behavioral model inference* (n=17). Moreover, Figure 3.6 shows that approaches based on *Alignment and matching* are often combined with approaches based on *Metric* or *Behavioral model inference*.

In this section, we present the six identified categories of approaches, and briefly describe representative example papers. We then continue with discussing the hybrid solutions.

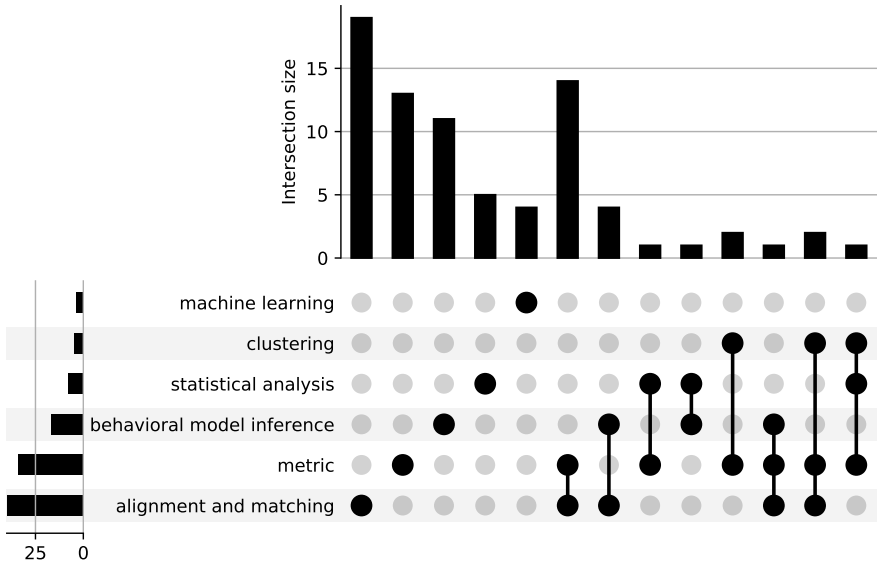


Figure 3.6: Number of (hybrid) approaches

Table 3.9: Categories of approach of log comparison

Method (#Papers)	Papers
Behavior model inference	
State machine inference (8)	[19, 36, 43, 121, 135, 256, 284, 294]
Invariant, predicate and pattern inference (10)	[89, 143, 200, 209, 242, 294, 315, 338, 398, 401]
Alignment and matching	
String alignment (25)	[17, 36, 98, 104, 141, 142, 145, 147, 163, 253, 266, 269, 271, 276, 310, 314, 327, 351, 369, 399, 400, 402, 403, 420, 440]
Graph alignment (12)	[14, 15, 36, 50, 73, 135, 263, 283, 362, 363, 396, 441]
String-graph alignment (4)	[27, 121, 256, 294]
Metric	
Similarity and distance metric (28)	[14, 15, 17, 36, 42, 73, 97, 104, 141, 142, 145, 147, 172, 231, 268, 269, 270, 276, 307, 326, 367, 369, 375, 396, 399, 402, 420, 444]
Importance and relevance index (6)	[16, 106, 151, 173, 442, 449]
Statistical analysis (8)	[43, 47, 78, 245, 302, 328, 367, 375]
Clustering (5)	[73, 97, 231, 269, 367]
Machine learning (3)	[152, 155, 183, 417]

Behavioral model inference. As found in our previous study (Chapter 2), developers often manually sketch models based on logs to obtain abstraction for comprehension and analysis. The papers that belong to this category present techniques that apply model inference techniques and perform comparison at the level of models. The identified differences can be linked to the graphical representations or the properties of system behavior. The models identified in these papers are classified into state machine models, and invariants, predicates and patterns. The state machine models can represent the *ordering information*. The majority of the papers in this category present techniques based on the model inference algorithm Ktails [52] which extracts a behavioral model from a set of execution traces, based on the set of sequences of k consecutive events found in the traces. Amar et al. [19] develop a comparison technique, so-called k-diff, which identifies sequences of length k or less that appear in one of the two logs. The sequence differences are visualized in the inferred models. The parameter k is configured by users to decide the length of sequence differences that they are interested. Another existing example can be found in the work by Ohmann et al. [284] where models are learned from logs and then resource utilization information is used to differentiate behaviorally similar executions that differ in resource consumption by detecting property violations. Invariants, predicates and patterns can represent various type of information such as *value of variables*, *ordering information*, *frequency of events*. An example can be seen in Lam et al. [200] that mines predicates that evaluate the state of the method call at different points in execution. The two sets of predicates inferred from two logs are then compared to narrow down the root cause of flakiness. By abstracting log information for comparison, techniques based on this category may reduce

the information to be compared, and help developers interpret the results with available models. However, the effectiveness of techniques depends on the level of details preserved. If important log details are abstracted away, identifying important differences between executions becomes difficult.

Alignment and matching. In this category, alignment and matching algorithms are adopted or developed for comparison. There are three types of alignment algorithms: string alignment, graph alignment and string-graph alignment. The string alignment algorithms consider logs as string sequences and align two logs event by event. *Ordering information* of logs is being compared in alignment. Existing alignment algorithms borrowed from information theory and bioinformatics have been used in log comparison. For example, algorithms for identifying the longest common subsequence (LCS) are seen in papers [163, 266, 314, 351]. Alignment approaches often suffer from scalability issues. As discussed by Hoffman et al. [163], the algorithm for computing the LCS requires a huge amount of memory and time when applied to longer logs. There are several ways to mitigate this problem. One of the ways, as presented in the work from Hoffman et al. [163], is to compress and categorize the information shown in log sequences before applying LCS. Another way, as demonstrated in literature, is presenting logs with graphs such as program dependency graph [362, 441], call tree [263, 363], state machines [396] and timeline graph [50, 283] and align the obtained graphs. Although the graph representation of logs abstracts some unnecessary details away, graph alignment still can suffer from the scalability issues. For example, as reported in [263], the computation of the edit distance of two trees could be expensive. Another problem that alignment algorithms often neglect is that they do not take log semantics into account. For example, LCS may blindly correlate neighboring entries in one log with entries very far apart in the other log [163]. To mitigate this problem, two studies [362, 363] adopt an execution alignment technique, so-called execution indexing [418], where call tree [363] or dependency graph [362] are extracted from traces and compared. However, it is machine undecidable whether the point in one execution corresponds to a given point in another execution (i.e., establishing matching points in two executions). Different from the traditional alignment algorithms (e.g., LCS), execution indexing aims to establish meaningful correspondence among points across multiple executions based on program structure and state, which reduces the risk of misalignment. The same challenge is also faced in graph alignment. As discussed by Walkinshaw et al. [396], comparing two state machines involves establishing correspondence of states and transitions in two state machines. One way to establish this correspondence is using state labels that are usually missing in the learned models from logs. The correspondence has to be established by taking the surrounding states and transitions into account (i.e., if two states are considered as a pair of matching states, then their surrounding states and transitions should be similar), which can become very expensive. The authors developed a state machine comparison algorithm which establishes the correspondence and computes the structural differences of two state machines efficiently. As shown in the evaluation of this algorithm, the accuracy of comparison could be affected by the extent to what one state machine is different

from another. Another group of alignment techniques aligns a string (i.e., a sequence of log events) against a graph (i.e., a graph representation of logs). An example can be seen in [27] where deviations between a log and a process model are identified by replaying the sequences of log events on the process model [215]. Through the literature review, we identified that the challenges of alignment techniques lie in establishing meaningful and precise alignment in a scalable manner. The techniques that take the semantics of log statements into account can potentially mitigate the problems. This also requires alignment techniques to deal with interleaving events introduced by non-determinism. That is, if event interleaving is not taken into account by alignment techniques, there is a risk of misalignment.

Metric. In this category, metrics are defined to quantify to what extent logs are different from each other (e.g., similarity and distance metric) or how important a trace statement it is compared to others (e.g., importance and relevance index). Existing metrics that quantify the edit distance between two strings are adopted in log comparison literature. For example, Levenshtein, Hamming and Minkowski distances are seen in several studies [142, 145, 276] to quantify the similarity of log sequences. Another example can be seen in Mirgorodskiy et al. [270] where execution time of function calls is extracted from traces of two hosts and represented using vectors. Manhattan distance is then used as a measure of behavioral distance between two hosts. If the behavioral distance between two hosts is small, each function will consume similar amounts of time on both hosts. The metrics from the field of information retrieval are borrowed to quantify the similarity between executions [307, 396]. For example, in [396], the distance between two sets of traces that represent the behavior of two executions are quantified using Precision, Recall, F-Measure, Specificity and Balanced Classification Rate (BCR). Precision and recall metrics provide the proportion of traces exercised in one execution but not in the other, quantifying to what extent two executions are different from each other. Instead of quantifying the similarity between two traces directly, six studies propose to use importance and relevance metrics that quantify how important a trace statement it is compared to others. Typically, in these studies, traces are generated from a set of passing and failing executions similarly to debug the failed executions. The importance of a trace statement is evaluated based on the occurrence and frequency of the trace statement in the passing and failing executions. For example, Hao et al. [151] propose a metric which is based on the intuition that statements mainly appearing in failing execution are viewed as highly suspicious. The advantage of these metric-based approaches is that it may give a quick idea of how similar the executions are, which could be useful when comparing a set of execution traces with each other. However, metrics might not be intuitive for developers to interpret the differences. The decision of which metric best reflects the similarity or dissimilarity requires more research efforts, to align with developers' perceptions [172].

Statistical analysis. In order to identify significant differences, some researchers consider log comparison as a statistical problem, using existing mature methods in the field of statistical analysis. Various statistical tests

are used to examine how different the distribution of execution time and frequency of events across multiple logs are. For example, Lu et al. [245] analyze sampling distributions of tasks' response time and execution time in traces using a proposed algorithm based on the non-parametric two-sample kolmogorov-smirnov test. The advantage of statistical approaches is that they help developers identify significant behavior variations, which may help developers prioritize their inspection. However, statistically significant differences are not necessarily meaningful differences [47]. It may hide the subtle but real symptoms of a problem. Moreover, they cannot explain why a difference occurs and what are the underlying problems.

Clustering. When dealing with a large set of logs, researchers apply clustering algorithms to categorize them based on their similarity and distance. That is, clustering approaches are naturally based on similarity and distance metrics. An example can be seen in the work by Lin et al. [231] where each log sequence is transformed into a vector and each event in the vector is assigned a weight that quantifies the importance of the event for problem identification. Cosine similarity between vectors (which represent two log sequence) is then computed. Having computed the similarity between every pair of log sequences, Agglomerative Hierarchical clustering technique [137] is applied to group the similar log sequences into clusters. Clustering approaches are suitable for comparing a large set of logs and can be used to reduce the number of logs that require a more detailed one-to-one comparison. The effectiveness of the approaches, however, depends on the choice of similarity metrics and underlying parameters of the used clustering algorithms (e.g., stopping criteria) [231, 367].

Machine learning. Machine learning algorithms are applied to train models that can classify execution logs for anomaly detection. He et al. [155] evaluate six machine learning algorithms including decision tree, log clustering, logistic regression, principal component analysis, invariant mining and support vector machine, aiming to provide guidance for developers to select proper algorithms. This work summarizes several challenges in log-based anomaly detection. First, the existing techniques extract event count from logs as feature. Other interesting features such as execution time of events and the ordering information are not considered. The reason that the ordering information is rarely considered is that it is challenging to extract a reliable ordering feature from logs which contain interleaving events. This challenge emphasizes the importance of identifying interleaving events in logs, which will be further discussed in Section 3.4.5. Second, these models may not provide intuitive insights, and developers often cannot understand what the anomalies are from log classification, which is a well-known research problem in machine learning - interpretability of results.

Hybrid solutions. Hybrid solutions have been identified in the literature, as shown in Figure 3.6. As we discussed, clustering approaches are by nature based on metrics, and metrics are often combined with alignment and matching

algorithms to compare the ordering information shown in logs. Interestingly, we observe that *behavioral model inference* is often combined with *Alignment and matching* as a hybrid solution. This hybrid solution is different from the techniques only based on *Behavior model inference* where comparison is integrated into the inference and the subsequent refinement process. Typically, this hybrid solution first infers a model from a set of logs and then aligns the model with either another set of logs (i.e., *String-graph alignment*) or a model inferred from another set of logs (i.e., *Graph alignment*). An example of the previous case can be found in [256] where a matching process between a trace and a finite state machine is developed on top of the Ktails extension mechanism. The trace is used to extend the finite state machine. All the extension points in the state machine are then considered as differences to be inspected by developers. Goldstein et al. [135] show an example for the latter case. In their work, finite state machines are inferred from logs with the Ktails algorithm and aligned to identify the groups of common, added and removed states. On top of *Behavior model inference* and *Alignment and matching*, *Metric* can be used to quantify the similarity between models. An interesting example can be seen in the work by Walkinshaw et al. [396] where state machines learned from logs are taken as inputs for graph alignment. In the alignment, states from two state machines are matched based on their similarity scores (i.e., common incoming and outgoing transitions of each pair of states). The precision and recall metrics are formulated to quantify the differences of state machines in terms of their structures and languages. A technique that combines *Behavior model inference* with *Statistical analysis* is identified in the work by Bao et al. [43] where not only the ordering information but also the frequency information is represented in probabilistic state machines. The work is based on Ktails algorithm, where important log sequences with a length of k that differentiate two sets of logs are identified with statistical tests.

3.4.3.5 Result Post-processing

As a result of log comparison, there may still exist a large number of differences. It could be overwhelming for software developers to inspect these differences and reason about them. Therefore, post-processing is often done to reduce the amount of information that developers have to analyze. We identified that filtering and ranking are the methods used in the literature.

Filtering is a process of removing irrelevant or less important information. Seven studies [271, 294, 315, 440, 449] define rules to reduce the number of differences presented to developers. For example, Mirgorodskiy and Miller [271] reduce the number of sequences that developers need to examine by merging the call sequences that share a prefix. Often, these rules are pre-defined by researchers with design knowledge of the techniques. For example, to remove duplications, Ranganath et al. [315] remove complex patterns and keep the simpler constituent patterns. Another example can be seen in the technique developed by Pastore et al. [294] where events that do not occur in every failure are removed. Several studies design metrics based on frequency of events [135, 367, 369] to quantify the importance of changing events and filter out the events that are scored lower than a certain threshold. The risk of applying metrics and pre-defined rules is that they might not align with developers' perceptions

and may require a threshold that could filter important log differences out. In addition to rules and metrics, slicing techniques [142, 400, 403] have been used in the literature to prune comparison results. For each trace difference, Tonella et al. [403] use dual slicing to construct a chain of trace differences by identifying control and data dependencies between trace differences. Instead of filtering out irrelevant information, some studies rank the comparison results based on heuristics. Similarly to filtering, metrics [14, 141, 151, 209, 270, 328] and rules [89, 246] are defined for ranking. For example, in the method developed by Alcocer et al. [14], components that present behavioral differences at each hierarchical level are sorted based on their execution time. Similarly, Luo et al. [246] rank modified methods based on the difference of execution time of these methods.

RQ2.3 summary:

Most log comparison techniques require *log-log* as input. We found that the *ordering*, *occurrence and frequency*, *resource consumption* and *value* are the information type compared most by log comparison techniques. The majority of techniques are based on the approaches of *Alignment and matching*, *Metric* and *Behavioral model inference*.

3.4.4 Evaluation Methods (RQ2.4)

In this section, we discuss the used evaluation strategies and participants involved in the evaluation of the log comparison techniques identified in the examined papers.

3.4.4.1 Strategy of Technique Evaluation

We employed a research strategy framework by Storey et al. [359] (explained in Section 3.3.5.4) to analyze the evaluation strategies used in the collected papers. The result is shown in Figure 3.7. Note that there are five papers applying a mixed approach (i.e., using more than one strategy). All papers contain a section of technique evaluation. Among them, 17 studies demonstrate their techniques with examples. In such papers, the expected effect of techniques is not or implicitly operationalized. The used examples are not necessarily a real-life, but often an artificial application. An example can be seen in the work by Maoz et al. [253]. The authors briefly describe their experience in using their techniques to visualize differences of execution traces for several applications.

Data strategies refer to evaluation methods that rely primarily on generated or simulated data. As the most frequently used method, they have been identified in 59 papers. The construct and operationalization are explicitly formulated and stated in these papers. The effects of techniques are evaluated with archival, generated or simulated data. Data strategies are often used to assess the performance of the techniques. An example can be seen in the work from Ramanathan et al. [314] where the overhead of instrumentation and log comparison is evaluated using a dataset collected from seven OSS projects (i.e., logs generated from two versions of these projects). Data strategies can also be used for assessing the accuracy of log comparison techniques for a certain software engineering task. An example can be seen in the work from Lam et

al. [200] where a dataset consisting of 44 flaky tests that belong to 22 software projects from 18 Microsoft products is used. For each test, the dataset contains 100 execution traces which are compared using the developed techniques to identify the root cause of the flakiness.

Seven papers have described the used lab strategies in which hypotheses are explicitly formulated and validated in highly controlled situations with human participants. Lab strategies are only used in the papers published in recent years; one paper in 2014, two papers in 2018, three papers in 2019 and one paper in 2020. For example, in the work from Alimadadi et al. [17], a controlled experiment conducted with 14 participants on a set of real-world comprehension tasks (e.g., understanding the addition of a new feature) is presented. The results show that using their technique helps developers perform program comprehension tasks 54% more accurately than other tools.

Two studies applied respondent strategies, with which developers were surveyed and provide feedback to researchers. An example from Pinto et al. [302] began with a data strategy and then asked for professional feedback on the results: "*We would like to verify if developers were aware of the performance issues that our approach has found. We collected feedback from eight developers through surveys for each target system.*"

The evaluation of techniques in a natural software development setting is referred to as field strategies. They have been identified in only two papers published in 2016 [231] and 2020 [50] respectively. Beschastnikh et al. [50] apply a mixed method which consists of lab, data and field strategies to evaluate a log analysis tool designed for comprehending distributed systems; 39 students are involved in a controlled experiment, and 70 students are involved in a field study where the computer science students used the tool for an undergraduate distributed systems course. Moreover, the tool has been evaluated by two researchers for debugging distributed systems that they are working on. Lin et al. [231] also apply a hybrid approach of data and field strategies. Their field study at Microsoft demonstrates the potential of applying the techniques in a real-world setting.

Next, we examine the mixed methods that have been applied. Our review identified several studies that used different combinations of strategies. For instance, we found a study that used data and respondent strategies [302], another that used data and field strategies [231], two studies that used lab and data strategies [19, 43], and a study that used lab, field, and data strategies [50]. While it is not yet a widespread practice, combining data strategies with other methods such as field strategies can add realism, lab strategies can provide control over influencing factors, and respondent strategies can enhance generalizability.

3.4.4.2 Participants

Nine studies involve human participants in their evaluation; seven studies [14, 17, 19, 27, 43, 50, 284] involve university students, three studies [14, 231, 315] involve professional software developers in the field, and one study [50] involves researchers. The involvement of both students and software developers can be seen in Alcocer et al.'s [14] study where 8 postgraduate students, 3 software developers and 1 undergraduate student are recruited in a lab

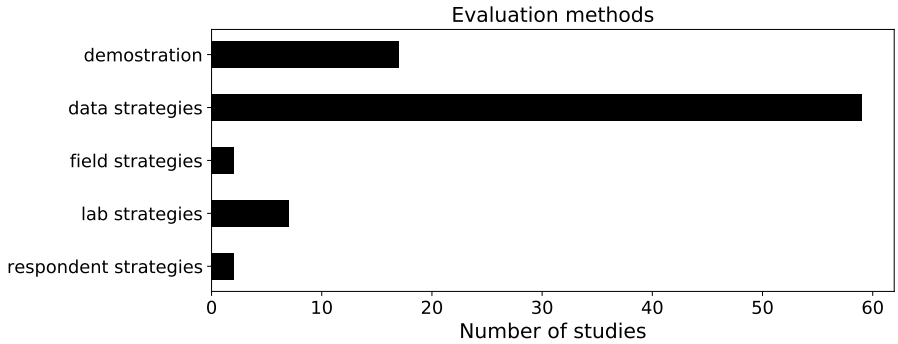


Figure 3.7: Used evaluation methods

experiment. There are two papers involving human participants, while no background information of the participants is explicitly provided. For example, Ohmann et al.[284] recruit a total of 10 students and 3 additional participants to voluntarily perform tasks in a lab experiment. The background of these 3 additional participants is not explicitly provided.

RQ2.4 summary:

Data strategies are the most frequently used method for evaluating log comparison techniques. Field and lab strategies are used few times in recent years. Data strategies have been combined with other strategies to increase quality of evaluation, although it is not yet a common practice. Most human participants in the evaluation are students.

3.4.5 Industry Challenges (RQ3)

In this section, we present how log comparison techniques from the literature address the industrial challenges presented in Chapter 2, i.e., enabling multi-abstraction comparison, handling interleaving events, and using additional information. There is only one study [17] that explicitly addresses all these challenges. Table 3.10 presents the papers that attempt to address at least one of the industrial challenges. It can be seen that the majority of studies (21 out of 25) develop log comparison techniques for issue analysis, requiring log-log as input. The most frequently compared information in these techniques are ordering and resource consumption (e.g., execution time). Most of the comparison methods are based on Alignment and matching. The commonality between these techniques in terms of activity, input, types of information and comparison methods suggests the possibility to combine or integrate different techniques to address the industrial challenges. Next, we discuss how these techniques address each of the challenges.

Table 3.10: Papers that attempt to consider at least one industrial challenge discussed in Section 3.4.5.

Paper	Input	Activities			Types of information					Comparison methods					Industrial challenges		
		Issue analysis	Comprehension	Verification	Value	Ordering	Resource info	Dependency	Frequency	Alignment	Statistical	Model	Metric	Clustering	Interleaving	Multi-level	Additional info
[294]	log-log	✓			✓	✓				✓		✓					✓
[78]	log-log	✓			✓						✓						✓
[14]	log-log	✓					✓	✓					✓			✓	✓
[449]	log-log	✓			✓								✓				✓
[338]	log-log	✓						✓				✓			✓		
[42]	log-log			✓		✓			✓				✓				✓
[200]	log-log	✓			✓	✓	✓					✓			✓		
[400]	log-log	✓			✓					✓						✓	
[402]	log-log	✓				✓	✓			✓			✓			✓	
[17]	log-log		✓			✓				✓			✓		✓	✓	✓
[440]	log-log	✓			✓	✓				✓					✓		
[403]	log-log	✓			✓	✓		✓		✓					✓		
[16]	log-log	✓					✓		✓				✓				✓
[310]	log-log	✓			✓	✓				✓							✓
[163]	log-log	✓				✓				✓						✓	
[369]	log-model	✓				✓				✓			✓			✓	
[327]	log-log	✓				✓		✓		✓					✓		
[302]	log-log	✓				✓	✓				✓						✓
[269]	log-log			✓		✓			✓	✓			✓	✓		✓	
[98]	log-log	✓		✓	✓	✓				✓					✓		
[328]	model-model	✓				✓	✓		✓		✓				✓		
[106]	log-log		✓			✓							✓		✓		
[15]	log-log	✓					✓		✓	✓			✓				✓
[271]	log-log	✓				✓				✓					✓		
[50]	log-log	✓	✓			✓				✓					✓		

3.4.5.1 Enabling Multi-level Comparison

Comparing logs at different levels of abstraction and guiding developers to drill down to differences at low level details with visualization are envisioned by the developers interviewed in our study presented in Chapter 2. We identified seven papers [14, 17, 163, 269, 369, 400, 402] that present techniques to allow comparison at different levels of abstraction.

The advantages of a multi-level comparison are twofold. First, it eases computation by considering more compact artifacts at higher levels of abstraction (i.e., keeping running times and memory requirements reasonable) [163, 269]. Second, multi-level log comparison with information linked at different levels can assist software developers in understanding the differences between software executions from a high-level of abstraction to the low-level details with the ability to identify and focus on particular parts of the differences individually. As an example that provides the first advantage, Miranskyy et al. [269] develop a log comparison technique that compresses traces into multiple levels of abstraction to speed up computation. These traces are compared iteratively from the highest level (i.e., set of function calls) to the lowest level (i.e., a sequence of function calls). The conjecture is that if two traces are dissimilar at a high level of compression, they will also be dissimilar at the corresponding lower level of compression. For example, if two traces have different sets of function calls, then the sequence of function calls would also be different. In this work, the compression techniques at different levels are independent of each other as long as the distance between compressed traces can be measured with metrics. Therefore, the log information at these levels is not automatically linked to each other, which may make comprehension difficult. Another six papers [14, 17, 163, 369, 400, 402], however, explicitly state that the multiple-level comparison is performed to ease comprehension by preserving the links between higher-level and lower-level information.

In four studies [14, 163, 369, 402], abstraction is defined based on the structure of software systems (e.g., components, classes, and call stacks), providing a semantics-aware way to structure and compare traces. Weber et al. [402] implement a hierarchical display of the trace differences based on call trees, which facilitates understanding of trace differences for root cause analysis. A timeline is designed to show sequences of function calls over time, visualizing at which depth of call stacks a deviation occurs between two executions. A similarity metric is calculated over time, allowing developers to easily detect areas with high dissimilarity. Moreover, a performance view that compares the duration of events from two executions is provided with a timeline. Alcocer et al. [14] help developers identify performance deviations across multiple executions with a hierarchical display of project, package, class and methods as well as the caller-callee relations between methods. With the developed tool, developers can easily identify code and performance changes at different levels of granularity, drilling down to the root cause of performance issues with the available link between code and performance changes. Tak et al.[369] develop three views for problem diagnosis of cloud applications. An overview is provided with a dashboard that shows the trend of log volumes, list of requests issued by the users, request types and list of log entries collected in real time. A model differencing view shows how the failed user request deviates from the

reference model. In this view, developers can pinpoint which components and at which time frames deviations occur. By tracing down further, developers can inspect a log differencing view, where deviations and potential problematic log lines are highlighted. Hoffman et al. [163] propose to abstract traces into several views that naturally arise in object-oriented programs (e.g. objects, methods, threads, etc.), and then compare the view-based traces. Views are linked among each other to capture program semantics in a scalable manner by identifying the common execution points in these views.

Different from abstracting based on the structure of software systems, Alimadadi et al. [17] introduce a new approach to software system abstraction based on configured tolerance of small variations. They define motifs, which encode entity sequences and allow high-level behavior inspection while preserving low-level details when needed. This is achieved by tolerating small variations in different manifestations of each motif. By definition, motifs are composite entities, which can contain other (sub-)motifs as their members. The approach discovers hierarchical and temporal relations between motifs using an algorithm and displays high-level system behavior overview with the ability to zoom in for details.

3.4.5.2 Handling Interleaved Events

Notoriously, interleaving caused by concurrency often introduces difficulties in comprehending the behavior of systems. Analysis of log differences in the presence of interleaving requires extra efforts because developers have to know if a log difference indicates functional differences. Fu et al. [121] obtain sequential logs by splitting original logs generated from distributed systems by thread ID or request ID. However, as argued by Tak et al. [369] who also develop log comparison techniques for distributed systems, global IDs rarely exists: *The scope of IDs carrying the identity of requests, users, sessions, resources and credentials is limited to a subset of cooperating S/W components. Where the scope of one ID ends, there could be other IDs in the log line that carry over the identity of current request processing.* The authors proposed a so-called log correlation algorithm, which identifies the relation between observed IDs. Unfortunately, the algorithm is omitted by the authors due to page limits.

Table 3.11: Methods of handling interleaved events

	Methods	**Ref**	**Limitations**
Logging	Partial ordering relations using logical clock	[50, 106, 338]	High performance overhead
	Variables that represent program states	[98, 440]	
	Memory allocation or/and thread activity	[327, 403]	
Domain knowledge	Pre-defined events that are interleaving	[200]	Low scalability
	Defined rules that distinguish request flows	[271]	
Heuristic	Configured thresholds that tolerate small variations	[17, 328]	Low accuracy

Essentially, the way to deal with interleaving is abstracting it away. We identified 11 studies which take interleaved events into account while developing their log comparison techniques and reported their solutions. Table 3.11 summarizes the methods used to identify interleaved events and their limitations. Seven studies log additional information that is used for abstracting irrelevant interleaving away. Three studies [50, 106, 338] use logical clock timestamps [113, 260] to capture the partial ordering of events. Using logical clocks (e.g., vector clocks and Lamport clocks), logical timestamps are generated for events in the system, and the causal relationship of events is determined by comparing those logical timestamps. However, capturing all partial ordering relations could be prohibitively expensive. Beschastnikh et al. [50] evaluate the overhead of applying logging framework XVector which implements vector clocks to capture interactions between hosts in a network system, and conclude that the overhead imposed by XVector makes it a feasible tool during development, but not in production. The authors therefore suggest investigating tracing frameworks X-Trace [119] and Dapper [343] that are designed for production use. Two studies [98, 440] log program states and use this information to determine whether the appearance of events in a different order leads a program to a distinct state. If this is not the case, the order of these two events does not indicate functional differences. The program state is represented by the values of a set of predefined variables. As discussed by both studies, the choice of what program variables to log and where to log them involves a trade-off. Capturing more variables would result in a more precise analysis, but introduce more logging overhead [98]. Different types of system may vary in the required number of variables to represent program state. For example, as discussed by Zhang et al. [440], for a typical client-side web application, the number of fields can be extremely large, resulting in a large performance overhead. Capturing partial ordering of events and program state is a generic solution for identifying different sources of nondeterminism.

We also identified two studies that focus on nondeterminism caused by memory address allocation or/and thread scheduling. Saissi et al. [327] track the order in which memory addresses are allocated and the order in which threads are spawned relatively to their spawning threads, and name them accordingly to achieve consistent IDs across multiple executions. The memory objects and thread identities are then abstracted across multiple logs. Similarly, Tonella et al. [403] obtain so-called lossless traces [208] which contain a program's entire control flow, including loop iteration and thread spawning. A canonical order of execution statements is obtained by taking the structure and state of the software program into account via indexing.

Domain knowledge is used in the literature to address interleaving. Lam et al. [200] use a predefined set of methods that return non-deterministic values (e.g., *System.Random.Next*), and allow developers to remove and add methods that they expect to be interleaved. Mirgorodskiy and Miller [271] decompose a trace into a collection of per-request traces based on defined rules. Each per-request trace is user-meaningful and more deterministic than the original trace, easing the comparison of multiple traces. The defined rules can be application-independent or application-specific. For example, one of the rules is the communication-pair rule that dictates the causal relation between

pairs of matching communication events (e.g., send and receive events). The methods based on domain knowledge may require the intensive involvement of software developers, making it hard to handle complex and heterogeneous systems.

In addition, heuristics based on the concept of threshold are used to reduce the large number of differences caused by event interleaving. Although this kind of methods is easy to implement without consuming extra resources, it suffers from low accuracy. Sambasivan et al. [328] compare a faulty period of execution with a non-faulty period of execution to identify mutations that manifest the root cause of fault. A mutation is identified if the difference of number of user requests in the faulty period and the non-faulty period exceeds a certain value. As discussed by the authors, choosing an appropriate threshold is challenging - a value that is too small will result in more false positives (i.e., obtaining more differences that are irrelevant). Similarly, Alimadadi at al. [17] handle the variations introduced by interleaving events with a penalty mechanism that tolerate small alterations in the patterns mined from traces. The technique also provides a hierarchical display of patterns. Minor variations are not presented at a higher level of abstraction, but a lower level of abstraction. Although the hierarchical comparison and display with a tolerance of variations can help developers focus on major differences, it does not distinguish interleaving differences from functional differences, and would still require developers to decide the relevance of variations.

3.4.5.3 Using Additional Information

Apart from the information shown in logs, information from other sources have been extracted and used for different purposes. We identified that ten papers use additional information, including *results of software analysis* and *code repositories*.

There has been an ongoing research topic about answering questions of what-to-log, how-to-log and where-to-log in software instrumentation. We identified four papers that use information from other sources to generate logs for comparison. There are several challenges in generating logs for root cause analysis. First, developers need to identify the inputs that trigger the normal behavior of systems that could be compared with the failing execution. Symbolic execution, which analyzes programs to determine what inputs cause each part of a program to execute, is used in the literature to solve this problem. Zuddas et al. [449] use a guided symbolic execution technique [180] which generates both failing and passing executions that mimic the observed failure. Similarly, Qi et al. [310] generate passing executions by using concrete and symbolic execution to synthesize new inputs that differ marginally from the failing input in their control flow behavior.

Another challenge is that a set of code locations that can differentiate behaviors between passing and failing executions should be identified to minimize performance overhead of excessive logging. This is particularly essential for systems that have critical timing requirements (e.g., embedded systems). Zuddas et al. [449] use symbolic execution to compute suspiciousness values of code lines and select execution points with high suspiciousness values to monitor program behavior. Chilimbi at al.[78] apply static analysis to

identify coupling between suspicious program paths and other parts of code, reducing the number of functions instrumented. Pastore et al. [294] compute code differences and log methods that contain at least a modified line of code. Bao et al. [42] perform source code analysis which extracts log templates, which are then used for log parsing. Moreover, to segment a large log file into multiple execution traces, the code analysis generates the reachability graph to reveal the reachability relations for any two log messages.

After the comparison, it is important to help developers interpret log differences. One of the ways to explain log differences is to augment log differences with additional information. Linking log difference to code changes has been implemented by log differencing techniques [14, 15, 16, 17, 302]. For example, with the technique developed by Alcocer et al. [14], source code differences are shown as a pop-up window when hovering over a method that differentiates two executions as shown in logs. Pinto et al. [302] mine software repositories to identify the commit that introduces the differences shown in logs, which guides developers to look into the relevant commit and the corresponding code changes.

RQ3 summary:

Out of 81 papers, seven studies take the needs for multi-abstraction comparison into account; twelve studies explicitly discuss their methods of handling interleaving events; ten studies extract additional information from other sources including the result of software analysis and code repositories; only one study explicitly addresses all these three challenges.

3.5 Discussion and Implication

Table 3.12 summarizes the main findings and implications. In this section, we discuss them in detail.

3.5.1 Use Cases

As shown in Chapter 2 and other studies conducted in industry [44, 146], logs are typically used for analyzing software issues, verification and improvement, and comprehension. In particular, logs are often compared when a software regression occurs. In this use case, logs are generated from the failing and passing runs, and compared to localize the problems and identify the root cause of regressions. Moreover, logs are also compared for verification and improvement purposes; when a change is made to software, developers need to verify whether the behavior of software has been changed as expected. Logs are then generated from two subsequent revisions and compared to verify the changes. As shown in Section 3.4.2, these major use cases from industry are aligned with the log comparison literature; nearly half of the papers motivate their log comparison techniques with a use case of issue analysis, behavior verification and comprehension. The alignment between industrial needs and research focus indicates that the research efforts have been made to solve major problem scenarios from industry. Researchers are encouraged to continue

Table 3.12: Main findings of literature review

Use cases (Section 3.4.2)	**Implications**
(1) The most frequent use cases of log comparison presented in the literature are aligned with the use cases in practices identified in our previous study [427]. That is, issue analysis, verification and improvement, and comprehension are the main use cases identified in literature of log comparison techniques.	The research efforts on log comparison techniques for software engineering have targeted the major problem scenarios in the industry.
Methods (Section 3.4.3)	**Implications**
(2) *Alignment and matching*, *Metric* and *Behavioral model inference* are the most commonly used and combined methods in log comparison techniques. Each of them has its limitations, suggesting possible improvements (e.g., methods based on *Alignment and matching* might only be useful if interleaving of events is considered in the context of concurrent executions).	The limitations of these methods should be overcome by taking industrial challenges into account and proposing hybrid solutions.
Maturity of techniques (Section 3.4.4)	**Implications**
(3) We observed a high use of data strategies and a low use of field, respondent and lab strategies in the evaluation of log comparison techniques. Only nine studies involve human in their evaluation, and most of human participants are students.	The strategies that involve human participants in industrial settings should be applied to increase the realism of the evaluation of log comparison techniques.
Industrial challenges (Section 3.4.5)	**Implications**
(4) Only a small share of papers explicitly consider the challenges identified in our previous study in industry.	More research efforts should be made to address industrial challenges in log comparison.
(5) Only one study explicitly considers all the three challenges.	Researchers should consider ways of combining or integrating the techniques that address different challenges.
(6) An accurate identification of interleaved events relies on logging more information in source code, which might introduce performance overhead.	More research is required to investigate the methods of capturing partial ordering relations between events and their impact on performance of systems.
(7) Logs can be abstracted in multiple ways, such as based on the structure of systems, or the tolerance of small variations.	The effectiveness of these abstraction methods should be evaluated to understand which abstraction methods can provide developers the most insights by taking different contexts into account. We conjecture that semantic-aware abstraction methods are essential for comprehension, and they can be combined with other abstraction methods to address the complexity of behavioral differences.
(8) Information from source code and software repositories has been integrated by several studies to help developers comprehend log differences.	To effectively utilize and extract additional information from other sources, an in-depth study is required to study what additional information developers use while inspecting log differences for their tasks in practices.

improving techniques that help with issue analysis, behavior verification and comprehension. Our finding about using log comparison techniques for researching developers' coding behavior [27] is interesting. This suggests the potential use of log comparison techniques in other SE comparative studies (e.g., junior and senior developers' bug fixing practices).

3.5.2 Methods of Log Comparison Techniques

As shown in Figure 3.6, most log comparison techniques are based on *Alignment and matching*, *Metric* and *Behavioral model inference*. Several hybrid solutions, such as the combination of *Metric* and *Alignment and matching*, have been proposed to overcome the limitations of the individual method. Apart from combining different methods, we suggest researchers to extend their techniques by taking industrial challenges into account. First, when developing comparison techniques based on *behavioral model inference*, choosing the right level of abstraction is essential for easing the computation of log differences for a computer program and the comprehension and interpretation of log differences for humans. However, it is still challenging to avoid missing important log differences. In the field of model inference, creating the right level of abstraction is still challenging [259]. To address this challenge, domain knowledge has been injected manually or automatically in some model inference techniques. Mashhadi et al. [259] develop a technique that allows users to iteratively define abstraction levels for the learned models used for debugging. It allows the user to create different models from the same set of execution traces, making a high-level state machine for initial understanding of the context and then zooming-in to the defective area by selecting the set of variables and function calls. Hooimeijer et al. [166] construct multi-level state machine models from logs by using knowledge of the software architecture, its deployment and properties. The multi-level state machine models inferred by these techniques can be further used for comparison. Indeed, comparing models at a single abstraction level may miss important behavioral differences. As favored by the software developers interviewed in our previous study, log information should be presented and compared at different levels of abstraction (as discussed in Section 2.5.4.3). It can be seen from Table 3.10 that the techniques based on behavioral model inference do not provide multi-level comparison, suggesting the need for extensions of these techniques. An example can be seen in a recent work (published after the studied papers have been collected) from Hendriks et al. [159] where a methodology is proposed to compare logs. In this methodology, multi-level state machine models that represent the behavior of multiple executions are obtained using the model inference technique developed by Hooimeijer et al. [166], and then compared using a model comparison technique developed by Walkinshaw et al. [396] which is studied in this literature review. By applying the existing model inference and model comparison techniques, this methodology allows developers to inspect execution differences in a top-down manner, supporting the localization of software components that behave differently in multiple executions, and the further investigation of behavioral differences of the components in detail.

In the presence of concurrent executions, approaches based on *Alignment and matching* might only be useful if the dynamic nature of complex systems

is taken into account (see discussion in Section 3.4.3). This idea is aligned with developers' experience with text-based comparison tools as discussed in Chapter 2. For example, when using text-based comparison tools for understanding functional differences of two software versions, a lot of interleaving events caused by concurrency are highlighted, overshadowing important differences related to functional changes. This alignment emphasizes the importance of explicitly dealing with interleaving events in log comparison. It can be seen from Table 3.10 that 8 out of 40 studies based on *Alignment and matching* [17, 50, 98, 121, 271, 327, 403, 440] have taken interleaving of events into account. We suggest researchers to take interleaving into account while using approaches based on the concept of *Alignment and matching*.

Approaches based on *Metric*, *Clustering* and *Statistical analysis* may lack an intuitive explanation of underlying problems indicated by a single value or resulting clusters. Additional information can help developers interpret results produced by log differencing techniques. As shown in Table 3.10, seven out of 41 studies based on *Metric*, *Clustering* or *Statistical analysis* [14, 15, 16, 17, 42, 78, 449] use further information in addition to logs for the developed log comparison techniques. Among these studies, four [14, 15, 16, 17] extract code changes to augment the resulting log differences.

3.5.3 Maturity of Log Comparison Techniques

In Section 3.4.4, we study the maturity of log comparison techniques by identifying their evaluation strategies and participant groups. We observe a high use of data strategies and a low use of field, respondent and lab strategies. Moreover, the mixed methods are also rare in log comparison research. The data strategies are often used to assess the performance (e.g., consumed time and memory) of the techniques. The rare use of field, respondent and lab strategies suggest that the effectiveness of techniques in assisting developers in their analysis tasks (e.g., software comprehension) might not be thoroughly evaluated. It is important to study how techniques could be applied in a real setting (e.g., with field strategies), how developers use or interact with the provided techniques (e.g., with field, respondent or lab strategies), and whether the techniques are better than other techniques in providing actionable insights for developers (e.g., with lab strategies).

This finding is not exclusive to log comparison research but aligned with general software engineering research, as shown in a study from Storey et al. [359] where 151 papers from two top SE venues are examined and the majority of papers adopted data strategies. It is not surprising that data strategies are more common than other evaluation strategies in log comparison research. Data strategies are suitable for evaluating some aspects of techniques, such as performance and scalability [359]. Moreover, data strategies can contribute to high generalizability of the findings if logs from various types of systems are used in the evaluation. The lack of field, respondent and lab strategies indicates the lack of human participants in the evaluation, threatening other quality criteria such as realism and control over human factors. In practices, log comparison is, however, an activity performed by humans. Human involvement in the evaluation is especially important for the techniques that promise to help developers ease the analysis activities. As shown in Section 3.4.3,

only nine studies have involved human participants, and the majority of participants in these studies are students. The difficulty of adopting field strategies is well-recognized in software engineering research [359]. To attract developers' and companies' interest in providing a study context and subject, we suggest researchers to propose a demanding and common use case (e.g., analyzing software regression and flakiness) for evaluation. Prioritizing and selecting the use cases of companies' interest is considered as the best practice to involve companies and ease the communication with them [129].

3.5.4 Addressing Industrial Challenges

In Section 3.4.5, we discuss to what extent the existing log comparison techniques take the identified industrial challenges (identified in Chapter 2) into account. With the goal of bringing techniques which are practically useful, the industrial challenges should be explicitly considered in the research work, as otherwise, the effectiveness of techniques cannot be validated. To answer our RQ3, we examine whether these log comparison techniques address the identified industrial challenges (i.e., providing multi-level comparison, handling interleaved events and augmenting log differences with additional information). Our results show that 25 of 81 studies (that we studied for RQ2.2-5) considered and explicitly handled at least one of the industrial challenges. Moreover, only one study considered all three challenges.

3.5.4.1 Identification of Interleaved Events

As presented in Table 3.11, there are multiple ways to handle the interleaved events. The methods based on logging additional information are more accurate, but introduce performance overhead to the system [50, 98, 440]. A trade-off is involved when using the methods that rely on logging. For example, when using the logged variables to distinguish program states, the accuracy of identification is higher when more variables that characterize program states are logged. That is, the methods based on logging additional information have to face the common challenges in software logging: what-to-log, where-to-log and how-to-log. In this study, we limit our investigation to the log comparison techniques that explicitly handle interleaved events.

We suggest researchers to study the large body of software logging methods: What are the existing logging methods that capture the causal relations between events? How accurate are the existing methods for the identification of interleaved events? How much performance overhead do the required logging methods introduce to the systems? Various tracing systems have been developed to track requests in distributed systems where concurrent executions inevitably occur, such as Zipkin [448], Canopy [187], X-Trace [119], Dapper [343], and Jaeger [177]. We suspect that the trade-off between performance of systems and accuracy of interleaving identification has to be investigated for different types of applications that have different system properties. For example, for embedded systems which often have strict timing requirements, one may have to sacrifice accuracy to ensure system performance. We suggest researchers to investigate the trade-off between the accuracy of identifying interleaved events and the introduced performance overhead for different types of applications.

3.5.4.2 Abstraction Methods for Multi-abstraction Log Comparison

We identified seven papers that compare log information at different levels of abstraction. As discussed in Section 3.4.5.1, researchers define abstraction of log information based on the structure of software systems or the tolerance of small variations.

We suggest researchers to focus on two directions. First, there are multiple studies [14, 369, 402] that abstract logs based on software structure. A more in-depth study that involves developers is necessary to identify which way of abstraction provides most insights. To evaluate the existing solutions, one can consider performing controlled experiments requiring developers to solve a certain software engineering task (e.g., to identify the root cause of flaky tests) with different log comparison techniques that provide a multi-abstraction comparison.

Second, researchers can consider combining different abstraction methods. According to an interview study conducted by Levy et al. [221] about comprehending large scale software, the key to understanding larger volumes of code is understanding abstractions and concepts. System understanding relies on the structure (architecture) such as the components, their connections, data flow, and also the design decisions, which is detached from low-level code details required by program understanding. The comprehension strategies are aligned with our study presented in Chapter 2 about log comprehension for large scale software. Hybrid solutions that combine or integrate multiple techniques can be proposed to provide multi-level comparison. For example, the technique developed by Tak et al. [369] can show how the communication patterns between software components differ in two executions, which help developers target the components for the inspection of internal low-level execution details (i.e., how the function sequences executed by a certain component differ in two executions). The differences in internal behavior of each component in the system can be obtained by integrating the work from Alcocer et al. [14] where a so-called performance Evolution Matrix is developed to hierarchically show the internal structure of software components (packages, classes, and methods) and the dependencies between these components. The differences that lie in the execution time and dependencies of methods are then visualized for multiple software versions, with the changes identified in source code. By combining the techniques from Tak et al. [369] and Alcocer et al. [14], both the communication patterns that represent high-level system behavior, and the performance and internal dependency that represent local behavior of each host can be hierarchically compared. Moreover, it could happen that there exists an overwhelming number of low-level differences. In this case, the method that infers the abstractions based on the tolerance of small alternations [17] can be applied to hierarchically display the internal differences for the component under inspection.

3.5.4.3 Additional Information for Comprehending Log Differences

As shown in Section 3.4.5.3, ten studies use further information from additional sources. Among them, five studies augment information from source code and software repositories to explain the identified log differences. However, it remains unclear what kind of additional information is useful and effective for

developers to interpret log differences. One way to investigate this question could be conducting observation studies to learn how developers compare logs when performing a certain task (e.g., root cause analysis), what additional sources of information they use, and how they extract the needed information. Understanding developers' information needs from difference sources can help researchers develop techniques that can automatically extract needed information. Information needs of various software engineering tasks, such as code review [293], design meetings [162], and team collaboration [193], have been studied in various contexts. Tao et al. [372] conducted an empirical study on how developers understand code changes. By gathering quantitative data from 180 survey participants, the authors collect questions developers asked when comprehending software changes, and the possible sources to obtain the information. For example, the question:"*Is this changed location a hotspot for past changes? How many times has this location been changed?*" could be answered by mining software repositories, while the question: "*Which documentation is linked to this code change?*" could be answered by leveraging requirement tracing techniques. We conjecture that the questions that developers ask in inspecting log differences may overlap with questions asked in inspecting code changes. A systematic research study is required to collect the information needs of developers for interpreting log changes and map the information needs to the existing techniques that automatically extract the information.

3.6 Related Work

In this study, we focus on existing literature about log comparison. To the best of our knowledge, there is no systematic literature review focusing on the comparison of logs. However, there are some literature review studies about log generation and analysis. In this section, we discuss these literature review studies and compare our study against them.

There are various literature review studies that have focused on log analysis techniques for different maintenance tasks. For instance, some studies have examined log generation and preprocessing techniques, which are critical in determining the quality of logs. El-Masri et al. [107] surveyed log abstraction techniques that transform raw log data into high-level information by analyzing 17 papers. Similarly, Chen et al. [77] surveyed log instrumentation techniques by studying 69 papers, with a focus on logging approaches, libraries, and integration, suggesting improvements for software logging techniques.

Other studies have examined log analysis techniques for specific maintenance tasks, such as reliability engineering. For example, He et al. [154] provided a comprehensive overview of automated log analysis techniques for reliability engineering by analyzing 158 papers. They focused on techniques that assist in three reliability engineering tasks: anomaly detection, failure prediction, and failure diagnosis. Similarly, Das et al. [90] examined the research trend of log analysis in failure prediction by examining 30 studies. Additionally, Svacina et al. [365] conducted a systematic literature review that examined recent trends in vulnerability and security log analysis, analyzing 34 papers and identifying the limitations of logging mechanisms in software systems. Our study focus on the comparison of multiple logs, which requires an input of multiple logs and

analyzes the differences between logs. Candido et al. [71] study 96 papers on logging techniques for software monitoring by examining work related to log engineering, infrastructure and analysis.

Our work differs from existing literature studies in several ways. Firstly, we concentrate on log comparison, which is based on our observation that developers frequently need to compare logs in their daily work (Chapter 2), whereas most of the previous literature studies focus on the broader use of logs. Different from some of the studies that focus on log analysis for specific maintenance tasks (e.g., [154]), we study what maintenance tasks that existing log comparison techniques attempt to assist. We identify that logs are frequently compared to analyze issues, which is consistent with the single log use case identified in the literature. This is also aligned with our previous finding (Chapter 2) that comparing logs generated from different software versions can often assist developers in identifying the root cause of regression problems. Secondly, we explore the evaluation strategies used to assess the techniques, emphasizing the lack of involving human participants in the evaluation of log comparison techniques. Moreover, one of our main contributions is that we evaluate the log comparison techniques against the industrial challenges identified in our interview study (Chapter 2), highlighting the need to improve these techniques to meet developers' expectations. In terms of the industry challenges we examine, a related systematic literature review [68] focused on techniques that match information from logs and stack traces back to source code. This review analyzed 16 papers and concluded that log-source matching techniques are primarily designed for fault localization, anomaly detection, and performance analysis. However, only two of the 16 studies reviewed combined additional information, such as bug reports and Stack Overflow, with logs and source code. This study addressed a research question similar to one of the industry challenges that we discuss in our study, where we investigate what additional information is used to aid in *log comparison*. We found that in our study, results of software analysis (such as symbolic execution) and source code are often combined with log information to interpret log differences. Despite our study having a different focus, we arrived at the same conclusion that additional information is rarely combined with log information by these log analysis and comparison techniques, which calls for more research into extracting useful information from other sources to help developers analyze logs.

3.7 Threats to Validity

This chapter provides a systematic overview of the state-of-the-art log comparison techniques, which is based on the analysis of 180 primary studies (81 SE related studies were analyzed for RQ2.2-5). In this section, we discuss the threats to the validity of this study.

3.7.1 Construct Validity

Construct validity is concerned with the extent the object of study truly represents the theoretical construct behind the study. First, the suitability of research questions determines whether the research objective can be addressed. To mitigate this threat, we took iterations to refine the research questions

through discussion. Furthermore, we formulated the questions related to industrial challenges, based on our previous qualitative study in industry. Second, the choice of keywords decides which set of papers are selected. We carefully select the keywords by studying the relevant papers about log comparison. We reduced the risk of missing papers that use keywords that mean the same (e.g., comparing and differencing) by querying the synonyms of these words. As explained in Section 3.3.1, we constructed a search query by concatenating keywords, which has a risk of missing papers that mention these keywords separately. We reduced this risk by performing snowballing. Third, we collected papers from selected data bases. There could be some relevant papers in other databases. We minimized the risk of missing relevant papers by performing one-round backward and forward snowballing. It could be that multiple rounds of snowballing are required to reach saturation. One-round snowballing is a practical solution in software engineering research. Fourth, the quality of papers may influence the results of our study. We select the primary papers based on the ranking of the venues, which reduces the risk of involving bad quality papers.

3.7.2 Internal Validity

Internal validity examines the cause-and-effect relationship between a cause and an outcome. In our study, the main threats to the internal validity arise from the limitation of the search engines. The databases are ever-changing, and constantly including more papers. Moreover, the underlying search algorithms could also be changing. This means that the same search queries may result in a different set of papers. However, with the performed snowballing, we consider our retrieval to be nearly complete to derive the classification schema discussed in this study.

3.7.3 External Validity

External validity is concerned with to what extent the results can be generalized. We collected the primary papers that are written in English. There could some relevant studies written in other languages. Another concern is that some studies might be described in gray literature (e.g., blog) or shared internally in companies. In this study, we focus on the resources that are easily accessible to a broad audience. A follow-up study could be studying the gray literature or performing a survey to collect the log comparison techniques developed in-house.

3.7.4 Conclusion Validity

Conclusion validity is a measure of the reasonable degree to which a research conclusion can be trusted. In our study, due to the large number of false positives we obtained from database search, we adopted automatic filtering based on the frequency of keywords in papers. There is a risk that we missed the papers that do not frequently mention the keywords but are still relevant. We mitigated this risk by randomly sampling 50 papers that are excluded by the automatic filter and manually validating their irrelevance. In the manual filtering process, two raters were involved for each paper, and the disagreements were resolved by the

third rater. Additional threats were introduced by using the ACM classification and by manual analysis (for RQ2.1). To limit the impact of the latter, we have used multiple raters and resolved the disagreement. The selected papers were analyzed only by the author of this thesis (for RQ2.2-5) due to the large number of papers studied in this work. The author of this thesis took iterations to examine information extracted from each paper and recheck the analysis while writing the paper.

3.8 Conclusion

In this study, we report on a systematic literature review of log comparison techniques. By combining keyword search and snowballing, we obtained 180 papers that are further classified into 6 categories based on the field of computing they belong to: most papers are related to software and its engineering (n=81) and information systems (n=63). By further analyzing papers related to software and its engineering, we identified that issue analysis is the main use case for log comparison in the literature. The ordering, occurrence and frequency of log events as well as the resource consumption and value information are the types of information compared most by log comparison techniques. The majority of techniques are based on the approaches of *Alignment and matching*, *Metric* and *Behavioral model inference*. The existence of hybrid approaches suggests possible ways to overcome the limitations of individual approaches. By examining the used evaluation methods, we found that data strategies, that rely on generated or simulated data, are most frequently used and combined with other strategies to increase quality of evaluation. Only nine studies involve human participants in the evaluation, and most human participants are students, indicating the need for increasing realism of the evaluation methods for log comparison techniques. Our previous study identified three industrial challenges of log comparison techniques: comparing log information at multi-level abstraction, handling interleaving events in logs, and extracting additional information to aid log comparison. We found that only a small share of techniques explicitly consider these industrial challenges.

Based on the result, we suggest researchers to further investigate approaches for addressing industrial challenges. For example, to handle interleaving events, we suggest a further investigation on the logging methods for capturing partial ordering relations between events, and their impact on the performance of systems.

4

Model Inference: Combining Active and Passive Learning

In chapters 2 and 3, we learned that developers and researchers use models to abstract log information. This observation is in line with the shift from code-based to model-driven engineering (MDSE) in the embedded industry. However, MDSE adoption faces the challenge of dealing with existing codebases. To address this, re-engineering activities, such as inferring behavioral models (e.g., state machines), are crucial. Model inference techniques can be classified as active or passive learning, but their practical application is hindered by limitations such as learning time and limited logs. In this chapter, we address RQ4 (highlighted in Figure 4.1), presenting a hybrid technique that extends active learning with execution logs and passive learning results. We evaluate the proposed technique on eighteen components used in ASML TWINSCAN lithography machines to infer models from the existing codebase and logs.

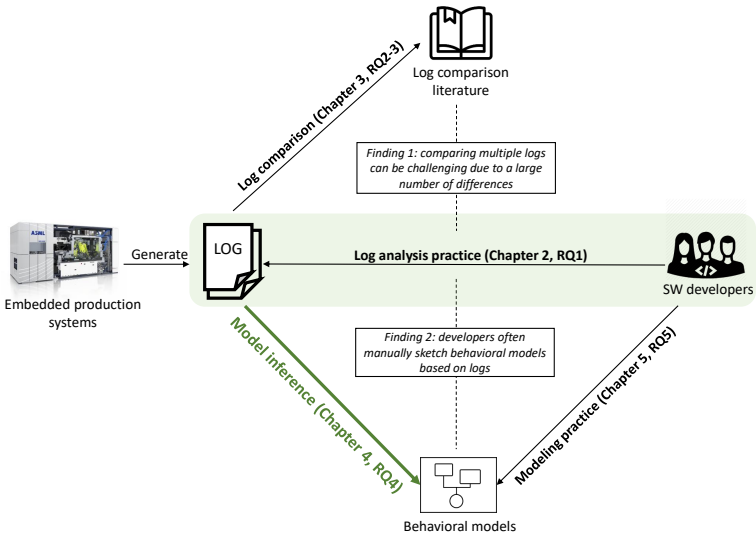


Figure 4.1: Research overview (RQ4)

4.1 Introduction

Unlike the traditional software development process, Model-Driven Software Engineering (MDSE) uses models as the main software artifacts. MDSE promises that presence of models will facilitate early verification of correctness and hence earlier defect detection, reducing development cost [136].

In order to benefit from the promises of MDSE, existing software systems have to be migrated. To tackle this problem, model inference techniques have been proposed in the literature. These techniques infer behavior using a running system, a so called SUL (system under learning), rather than modeling from scratch. The inferred models (e.g. state machines) can then be verified, simulated, transformed or used to generate new code.

Model inference techniques can be categorized into active and passive learning techniques. *Active learning* techniques [23, 174, 320] are based on the query-response mechanism. They iteratively interact with a running system by sending inputs (queries) and observing outputs (responses), and infer hypothesized models based on the interactions. Such techniques guarantee to learn the complete behavior under the assumption that the counterexamples differentiating the hypothesized model from the system can be found via conformance testing. However, as discussed by Vaandrager [383], the required number of test sequences grows exponentially with the size of the system. Executing such a large set of test sequences is very time-consuming. In practice, the learning process has to be stopped at some point. In such a case, one can never be sure whether the learned model represents the complete behavior of the running system. Hence, application of active learning in practice induces a trade-off between efficiency and behavioral coverage.

Passive learning techniques [52, 395, 406] infer models from a set of execution logs. Since the logs correspond to a limited number of use cases, the learning

results are also incomplete [134, 394]. Moreover, these techniques introduce overapproximation [65, 203, 238] making it hard to learn the complete behavior of the system.

To get a better understanding of how testing hinders active learning to scale in real settings, we conducted an exploratory pilot study at ASML, provider of lithography systems for the semiconductor industry. We applied active learning to 218 components from the TWINSCAN lithography machine and observed that when the total active learning time increases, the learning is dominated by the time spent in testing the correctness of the hypothesized models. This observation confirms the discussion of Vaandrager [383] that the required number of test sequences grows exponentially with the size of the system. Moreover, we also find that it is particularly hard to learn the complete behavior of systems where earlier choices restrict the behavior much later on. We name the early choice behavior problem as *far output distinction behavior*. Indeed, active learning requires counterexamples that distinguish the hypothesized model from the system, to achieve completeness. For systems with the far output distinction behavior, the counterexample is a long sequence of inputs capable of reaching the system states where different outputs can be observed. It requires a lot of time for the conformance testing algorithms to explore all possible input sequences of a certain length and find the counterexamples. The far output distinction behavior is one of the reasons why in our pilot study active learning could not finish learning all the 218 components within 1 hour. By inspecting these components, we find that the unlearned behavior occurs frequently during system execution. Artifacts created during system execution, such as logs, and subsequently passive learning results obtained from them, can thus be expected to contain this unlearned behavior. Hence, additional information derived from logs or passive learning results is expected to speed up finding the counterexamples, improving the efficiency of active learning.

Based on the pilot study we explore *whether active and passive learning can be combined to improve the efficiency of learning, while guaranteeing a certain minimum behavioral coverage*. From the passive learning perspective, active learning can be used to find the exceptional behavior that is not captured by the execution logs. For active learning, the logs and passive learning results can be used to learn the behavior efficiently. A certain minimum behavior coverage can be guaranteed; the observed behavior captured by the execution logs will be included in the learned result. To evaluate whether combining active and passive learning can improve the efficiency of active learning, we applied the combined approach to 18 components from 218 components of our pilot study. We observe that active learning finishes significantly faster and results in complete behavior. In particular, the combined approach helps to distinguish states that were hard to distinguish with the existing setup, without exhaustively exploring all combinations of input actions state by state.

The main contributions are the investigation of the scalability of active learning and the causes of time-consuming testing (the pilot study), and an improved active learning technique that integrates execution logs and results of passive learning.

Outline. After discussing the background in Section 4.2, we present the pilot study in Section 4.3. Then we introduce the combined approach in Section 4.4

and evaluate it with industrial components in Section 4.5. Finally, we discuss related work and conclude our paper in Sections 4.6 and 4.7.

4.2 Background

Most model inference techniques learn state machines. Several algorithms [23, 52, 174, 395] have been proposed over the years to implement active and passive learning. Since we focus on the conceptual weaknesses of active and passive learning, we introduce only the generic concepts underlying the algorithms. For a more detailed explanation of different algorithms, the reader is referred to the paper by Vaandrager [383] for active learning, and the one by Stevenson et al. [357] for passive learning. Below we introduce the necessary concepts and definitions that are used in this chapter.

4.2.1 State Machines

We choose Mealy machines to represent the results of model inference because Mealy machines provide the notion of input and output actions that are often used to model reactive systems.

Definition 1 — Mealy machine. A (deterministic) Mealy machine is a tuple $\mathcal{M} = \langle S, \Sigma, \Omega, \sigma, \lambda, \hat{s} \rangle$, where S is a set of states, Σ is a set of input actions, Ω is a set of output actions, $\sigma : S \times \Sigma \to S$ is a transition function, $\lambda : S \times \Sigma \to \Omega$ is an output function and $\hat{s} \in S$ is the initial state.

Definition 2 — Deterministic Finite Automaton. A Deterministic Finite Automaton is a tuple $\mathcal{DFA} = \langle S, \Sigma, \sigma, F, \hat{s} \rangle$, where S is a set of states, Σ is a set of input actions, $\sigma : S \times \Sigma \to S$ is a transition function, $F \subseteq S$ is a set of accepting states, and $\hat{s} \in S$ is the initial state.

Given a set of input traces, a prefix tree acceptor $\mathcal{PTA}$ is a tree-like $\mathcal{DFA}$ where each input trace in the set is represented by a path from the initial state to an accepting state, and no state has multiple incoming transitions.

Example 1 The Mealy machine in Figure 4.2 implements functions a_i and b_i with return values a_o and b_o, respectively. The $\mathcal{PTA}$ corresponding to the set of execution traces $\{a_i a_o a_i a_o b_i b_o b_i b_o, a_i a_o b_i b_o\}$ is shown in Figure 4.3.

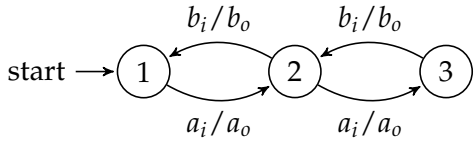


Figure 4.2: A Mealy machine of a SUL. The notation i represents a function call, while the notation o represents the return value of the function call.

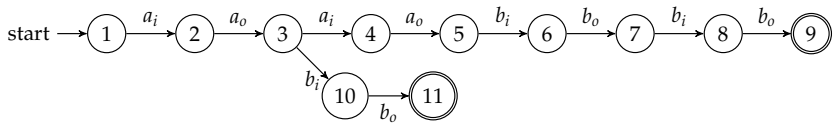
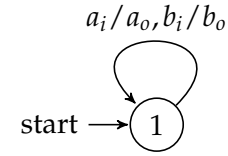


Figure 4.3: PTA for a set of traces $t = \{a_i a_o a_i a_o b_i b_o b_i b_o, a_i a_o b_i b_o\}$. The notation i represents a function call, while the notation o represents the return value of the function call

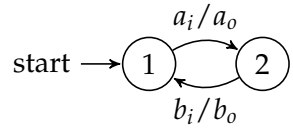
4.2.2 Completeness of Learning Results

Completeness requires that the learning results contain all the behavior that is allowed by the SUL and nothing more than that. Completeness can hence be violated by overapproximation or underapproximation. Overapproximation means that the learned model allows behavior that is not allowed by the SUL. Underapproximation indicates that some of the behavior of the SUL is absent from the learned model.

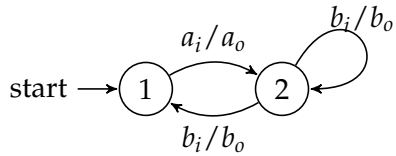
Example 2 The model in Figure 4.4(a) overapproximates the SUL from Figure 4.2. In fact, this model allows any sequence of inputs. The model shown in Figure 4.4(b) underapproximates the SUL. It misses the occurrence of input sequence $a_i a_i b_i b_i$ which is allowed by the SUL. The model shown in Figure 4.4(c) both overapproximates and underapproximates the SUL. This model allows input sequence $a_i b_i b_i$ which is not allowed by the SUL, while it disallows $a_i a_i b_i b_i$ present in the SUL.



(a) Model with overapproximation



(b) Model with underapproximation



(c) Model with overapproximation and underapproximation

Figure 4.4: Model (a) overapproximates the SUL from Figure 4.2, (b) underapproximates it and (c) both overapproximates and underapproximates it. The notation i represents a function call, while the notation o represents the return value of the function call

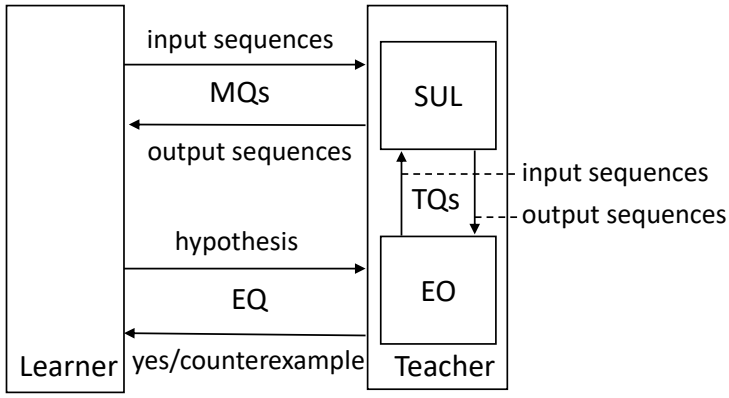


Figure 4.5: Active learning framework

4.2.3 Active Learning

In 1987, Angluin proposed the L* algorithm which implements a well-known active learning framework [23]. The original L* algorithm was designed to construct DFAs and was later adapted to learn Mealy machines, enabling learning for reactive I/O systems [255]. The active learning framework (Figure 4.5) assumes the presence of a *teacher*, which consists of the SUL and an equivalence oracle. It also assumes that the SUL can be represented by a regular language.

Initially, the learner only knows the set of input actions Σ and output actions Ω of the Mealy machine that represents the behavior of SUL. When learning, the learner iteratively executes two steps. In the *first* step, the learner asks membership queries (MQs) to the SUL to obtain sequences of output actions in response to sequences of input actions. For example, for the SUL represented in Figure 4.2, output sequence $a_o b_o$ is the response to input sequence $a_i b_i$. The learner then proposes a hypothesis model in the form of a Mealy machine, based on the output sequences. In the *second* step, the learner verifies the correctness of the derived hypothesis model by posting the hypothesis as an equivalence query (EQ) to the equivalence oracle (EO); the oracle composes test queries (TQs) which are sent to check the equivalence with respect to the SUL. A TQ, similar to a MQ, checks whether the system's response to an input sequence agrees with the response expected from the hypothesis. If there is a mismatch between the responses to the TQ from the hypothesis and the SUL, then the input sequence is considered as a counterexample. Based on the counterexample, the learner refines the hypothesis with further MQs. The learning process continues until the equivalence oracle cannot find a counterexample anymore to distinguish the hypothesis from the behavior of the SUL.

The learning algorithm guarantees the completeness of the learned model under the assumption that the equivalence oracle always returns a counterexample, given a counterexample exists. Peled et al. [296] proposed to use conformance testing to approximate the equivalence oracle. The partial W-method (Wp-method) [123] is a conformance testing technique that, given an upper bound m on the number of states of the target model, constructs a set

of TQs to find the difference between the hypothesis and the SUL. Hence, if such a bound m is known for the given SUL, the equivalence oracle based on the Wp-method guarantees to find a counterexample if one exists.

However, in practice finding m is non-trivial as it requires a comprehensive understanding of the SUL [347]. Furthermore, the number of TQs increases exponentially in $m - n$ where n is the number of states of the hypothesis [383]. Underestimation can cause incompleteness while overestimation can cause scalability issues. Limited by time, in practice one has to stop testing the hypotheses at some point, sacrificing completeness. The last hypothesis, proposed before stopping, can overapproximate the SUL, underapproximate it or both overapproximate and underapproximate it for different parts of the behavior.

4.2.4 Passive Learning

Different from active learning, passive learning algorithms learn a model based on a provided set of traces. The majority of passive learning algorithms are based on state merging [201, 380]. By merging equivalent states, the behavior is generalized (i.e., allowing more possible traces). However, a greedy merging strategy will over-generalize the allowed behavior in the learned models. One of the ways to avoid overgeneralization is to provide negative traces which provide counterexamples to invalidate aggressive state merges. Therefore, many passive learning algorithms, such as RPNI [285], expect both positive traces (i.e., traces accepted by the SUL) and negative traces (i.e., traces rejected by the SUL).

Below, we explain the concept of state merging with a well-known RPNI algorithm. Many algorithms were later developed on top of it. RPNI starts from positive traces to build up a PTA. Next, the algorithm iteratively merges pairs of states. Passive learning algorithms are often different in how they select a pair of states to merge. RPNI merges states based on the concept of quotient automaton. For more details about the concept, we refer to the original paper of this algorithm [285]. An important property of this merging strategy is that the resulting model is the superset of the original model (i.e., the resulting model accepts the language of the original model). State merging might cause non-determinism, which is then removed by merging additional states. For example, given the PTA in Figure 4.3, RPNI might decide to merge states 3 and 7, hypothesizing $a_i a_o (a_i a_o b_i b_o)^* b_i b_o$. Next, the validity of the merge is checked: merges that accept negative traces are disallowed to avoid poor generalization. This process continues until no further merges are possible. Upon termination, the algorithm has learned a model that accepts all positive traces and rejects all negative traces. The language of the PTA, representing the exact behavior of the set of positive traces, is a subset of the language of the model resulting from passive learning.

The passive learning algorithms guarantee to learn a complete model given a complete set of traces. Notions of completeness for a trace set differ for different algorithms: e.g., RPNI requires that the positive trace set visits every state and transition in the behavior of the SUL, and the negative trace set distinguishes every pair of states in that same behavior.

In practice, execution logs consisting of traces are used as the inputs to passive learning algorithms [239]. The execution logs usually only cover

limited use cases, and contain only positive traces [5]. This means that the negative traces are practically absent [239], although such traces are needed to avoid overgeneralization as proven by Gold [134]. In the absence of negative traces, heuristics are typically used to prevent over-generalization [201]. These heuristics can vary, but typically still lead to overapproximation of some parts of a system. Together with the incompleteness of the logs, the passive learning result presents the same drawback as the active learning result, that is, both overapproximation and underapproximation can exist in the learned models.

4.3 Pilot Study

As introduced in Section 4.2.3, active learning can suffer from scalability issues due to the number of TQs required for validating hypotheses. However, there has little empirical evidence that shows the scalability of the technique in practice. To better understand the scalability challenges of active learning and the role of the testing phase, we conduct a pilot study at ASML, provider of lithography systems for the semiconductor industry. Our approach to be presented in Section 4.4 is inspired by the findings of the pilot study.

Smeenk et al. [346] applied active learning to learn an industrial component and reported that they did not learn the complete behavior of the component, having used more than 263 million queries over a learning period of 19 hours. Their study evidently supports the claim that the low learning efficiency reduces the scalability of active learning. However, they did not study *(1) to what extent testing is the bottleneck in the active learning process* and *(2) which distinguishing behavior between hypothesis and SUL is time-consuming to find via testing*. Answering these two questions is a prerequisite to developing more scalable solutions. Hence, we design a pilot study to answer these questions.

4.3.1 Study Design

In this section, we discuss our study design.

4.3.1.1 Study Objects

To apply active learning as described above, we need to identify the upper bound on the number of states m. In general, this step requires profound knowledge of the SUL and precise estimation of the upper bound. Therefore, we opt for components that originally were developed using traditional engineering practices and later manually migrated to MDSE, while preserving the functionality. Since these components are MDSE-based, we can use the number of states from the behavior of the MDSE models (i.e., reference models) as m. Moreover, since the components were first developed in a traditional way, they can be expected to exhibit a level of control-flow complexity comparable to other components developed using a traditional software engineering approach. Based on these criteria, we select the logistics controller of ASML's TWINSCAN lithography machine. This controller is in charge of scheduling the logistical process within a wafer scanner, making sure that each wafer is processed according to a specified recipe. In 2012, it was manually redesigned using an MDSE technology called Analytical Software Design (ASD) [390]. Over the years, 28 developers have performed more than 1,500 commits to the master branch of the version control repository of the controller. The resulting software

consists of 218 communicating ASD components. Each component is modelled as a Mealy machine. The number of states in the Mealy machines varies between 1 and 18,229. The generated code consists of more than 700 KLOC.

4.3.1.2 Active Learning Setup

We opt for the state-of-the-art active learning algorithm TTT [174] together with the Wp-method for testing. The TTT algorithm improves L* algorithm by optimizing the internal data structure of the algorithm and reducing redundancies in counterexamples. As a result, the TTT requres fewer MQs to learn models. Active learning consists of two steps, learning or refining of the hypotheses using MQs and testing these hypotheses using EQs and TQs. For each component, we separately measure the learning time and the testing time. We run the learning process with a timeout of 1 hour. In case of a timeout, we consider the last hypothesis constructed by the learner as the learned model.

4.3.1.3 Model Comparison

Since we apply active learning to the components for which code is generated from models, we have the reference model to examine whether the learned models are correct or not. Following the comparison framework proposed by Aslam et al. [32], we consider a learned model to be complete if it holds a certain formal relation with its corresponding reference model. The used formal relation in this framework is called weak trace inclusion. According to the authors, this formal relation is used because the theory of active learning is based on traces. If a mismatch exists between the learned model and its reference model, we use the structure-based comparison method of Walkinshaw et al. [396] to identify the differences. We then analyze why the differences between the hypothesis and the SUL are hard to find via conformance testing, and what improvements might be beneficial to make this process efficient.

4.3.2 Results

In this section, we discuss the results of this pilot study, which applies active learning to the 218 components from ASML. We measure the total time of active learning, which includes learning time and testing time. Among the 218 components, 112 have been learned within 1 hour. For these components, when the total learning and testing time is small, learning is responsible for more than half of the total time of active learning (see Figure 4.6). However, as the total time of active learning increases, the ratio drops: when the total time of active learning exceeds 1 minute, the learning time is less than 3% of the testing time.

Next, we take a closer look at one of the remaining 106 components. The reference model for this component is a Mealy machine with 14 states and 144 input actions. In this model, we identify a pattern which cannot be learned by active learning within one hour.

We call this behavioral pattern as **far output distinction behavior**, as shown in Figure 4.7. Note that Figure 4.7 presents only the states that show the structural differences between the last hypothesis constructed for this component and the component itself (SUL). The reference model in Figure 4.7(a) shows that output actions c_1 and d_1 can only occur in the upper path, following the input action a_1 (as shown in bold). Similarly, output actions c_2 and d_2 can

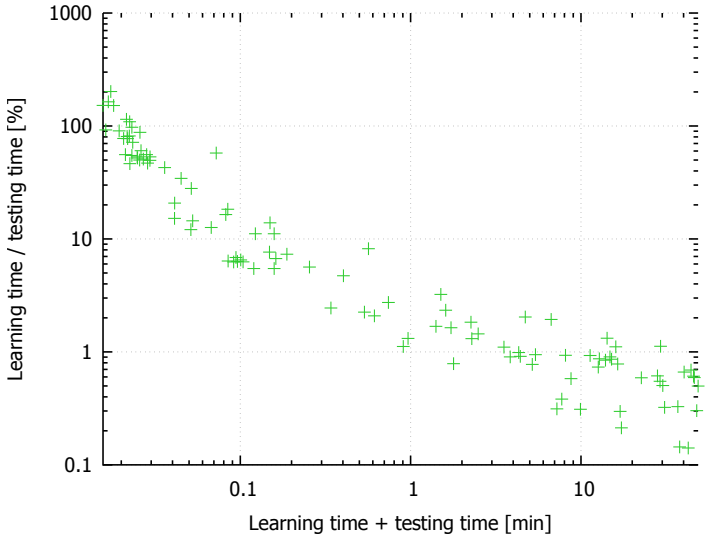
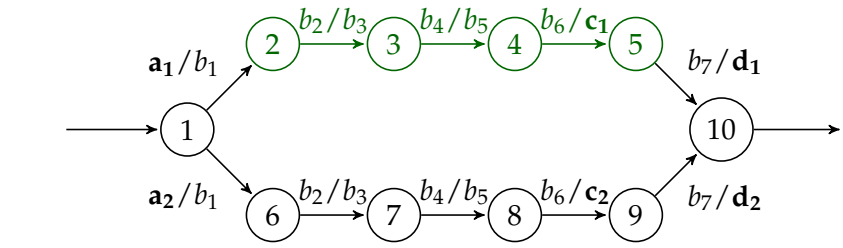


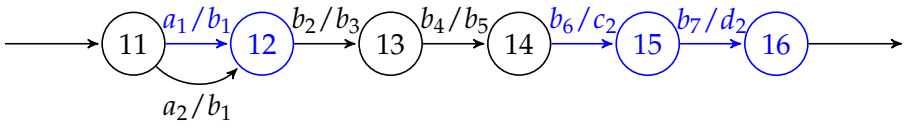
Figure 4.6: Learning time and testing time in active learning

only follow the input action a_2 in the lower path (also shown in bold). However, the learner only successfully learned the path starting with input action a_2, as shown in Figure 4.7(b). It also tried to explore input action a_1 from state 11, but failed to distinguish the paths because no different output action was immediately observed. As a result, output actions c_2 and d_2 can follow input action a_1 (as shown in blue). This means that the input sequence $a_1b_2b_4b_6$ was not explored, as otherwise the counterexample that distinguishes state 2 from state 6 would have been found. Searching for such a counterexample requires the equivalence oracle to test all combinations of the input actions, to explore the behavior from state 2 to state 5 (in green) with sequences that consist of 3 input actions. Recall that the component is much bigger than suggested by Figure 4.7, which focuses only on the structural differences between the last hypothesis constructed and the component itself; the component has 14 states and 144 input actions. Exploring sequences of 3 input actions requires exploring 144^3 (i.e., 2,985,984) combinations in the worst case, while only one of them is relevant and can refine the model. Furthermore, the equivalence oracle not only considers behavior between states 2 and 5, but between other pairs of states as well. Even with a timeout of 8 hours, those two paths could not be distinguished.

We inspected other unfinished learning cases. The far output distinction behavior often appears in the reference models. Discussing this observation with the developers of the logistics controller we learned that such behavior is common in the controller due to non-stop parallel processing in TWINSCAN systems: for maximum accuracy and productivity a TWINSCAN measures one wafer while imaging another one. Hence, the controller has to, for instance, schedule the movement of two chucks which hold the measured and imaged wafers respectively. This system requirement is reflected by different control



(a) Reference model



(b) Learned model

Figure 4.7: Example showing the far output distinction behavior problem in the testing part of the active learning process. The actions in bold distinguish two paths that are not learned by active learning. The states and transitions in green show a path that is not explored by testing, while the states and transitions in blue highlight the execution path disallowed by the reference model. The numbers shown in states are labels and do not suggest ordering.

outputs given similar input sequences in the components of the controller. However, the far output distinction behavior is typically missing from the learned models.

4.3.3 Conclusion

This pilot study confirms that testing is the bottleneck of active learning as previously discussed in literature [346, 383]. It also shows that most of the testing time is spent on finding counterexamples that distinguish the states differentiated by an output action on a (far) future state.

In the pilot study, we took the advantage of MDSE components for which the number of states is known, and hence the Wp-method can be suitably configured. However, in practice users usually know little about a legacy SUL that was developed using traditional engineering practices. The underestimation of the upper bound m removes the guarantees the Wp-method provides, i.e., the learned models might be incomplete, even though we run the learning till it ends. Given the difficulty of estimating the upper bound m and the large amount of required testing time, obtaining a model that is far from complete is not unusual.

Furthermore, Figure 4.7 suggests that if one can find the counterexample $a_1 b_2 b_4 b_6$ faster, then the model can be completely learned in a shorter amount of time. Since ASML developers recognize far output distinction behavior as part of their regular system behavior, we expect that artifacts produced during system execution, such as logs, capture this far output distinction behavior. Hence, next we design the *sequential equivalence oracle* extending the Wp-method

conformance testing to generate counterexamples based on execution logs and passive learning results.

4.4 Sequential Equivalence Oracle

We start by presenting the overall architecture of our sequential equivalence oracle which has been briefly sketched [30], and then focus on its individual components.

4.4.1 Architecture

The idea behind the sequential equivalence oracle is similar to the idea behind hierarchical memory in computer architectures, i.e., expensive operations are used only when necessary. In computer architectures, different levels of caches are serving as staging areas, to reduce the needs of visiting relatively slow main memory and disk, when the CPU searches for data.

We can compare the traditional active learning approach to a computer architecture without caches. In the traditional active learning approach, when the learner requests a counterexample using an EQ, the Wp-method searches for the counterexample by asking the generated TQs to the SUL. This is an expensive operation as shown in the pilot study. To reduce the frequency of this expensive operation, we insert "caches", i.e., cheaper oracles, into the active learning process, as shown in Figure 4.8.

Acting as the first "cache", the Log-based oracle starts searching for a counterexample when the hypothesis arrives. It returns a counterexample if found, otherwise the hypothesis is forwarded to a PL-based oracle. Similarly, the PL-based oracle returns a counterexample if it finds one, otherwise the hypothesis is forwarded to the Wp-method oracle. The role of the Wp-method oracle is comparable to that of main memory and disk where the data can always be fetched if it exists, at the price of time. We do not modify the Wp-method oracle; it works in the same way as in traditional active learning.

4.4.2 Log-Based Oracle

The Log-based oracle is based on the observation that logs represent actual behavior of the system. Hence, counterexamples can be found by identifying traces present in the log that cannot be generated by the hypothesis model H. To implement the Log-based oracle we collect execution logs for the SUL and construct a PTA, M_{PTA}, from these logs. Then, we compute the difference automaton for $M_{PTA} \setminus H$. If the resulting automaton has at least one accepting trace, which shows the language is not empty, the Log-based oracle generates a trace and returns it as a counterexample. Otherwise H is forwarded to the PL-based oracle.

4.4.3 PL-Based Oracle

Most passive learning algorithms ensure the inclusion of the input logs in their learned models. Hence, the execution logs are accepted *both* by the result of a passive learning algorithm and by the hypothesis H, that is, they are accepted by the *intersection* of the DFA representing the result of passive learning and H. The behavior represented by this intersection is more likely to belong to the

SUL than behavior represented solely by the result of passive learning (but not H) or solely by H (but not the result of passive learning).

The PL-based oracle is based on the *conjecture* that passive learning generalizes the logs, potentially overapproximating the behavior, and that the behavior learned solely by passive learning (but not H) might contain valid generalization that belongs to the SUL.

As opposed to the Log-based oracle, the PL-based oracle is based on a *conjecture*. This is why in addition to computing the difference automaton for the hypothesis DFA and the DFA representing the passive learning result, and generating a trace from that difference, we need to check whether the generated trace is a valid counterexample or not. The oracle posts the generated trace as a TQ to the SUL. If the trace is accepted, it is valid and should be included in the behavioral model, so the trace is sent to the learner as counterexample to refine the learning. If the trace is rejected, it is excluded from the passive learning result as invalid generalization. This process continues until a counterexample is found or all generated traces are examined. The hypothesis is then forwarded to the Wp-method oracle if no counterexample can be found anymore.

4.4.4 Implementation

The sequential equivalence oracle is developed on top of LearnLib [311], an open-source framework providing the implementation of several active learning and testing algorithms.

As our target model is a Mealy machine, we use the learning algorithms provided by LearnLib for learning Mealy machines. We choose Mealy machines to represent behavioral models because Mealy machines are a good representation for reactive systems as they fit seamlessly with function calls and return values. When implementing the Log-based and PL-based oracles, we first convert hypotheses represented as Mealy machines to DFAs, and only then compute difference automata. The worst case complexity of the conversion and subsequent DFA operations is $O(n^2)$, where n is the number of states of the Mealy machine. Since LearnLib does not include means of computing the difference between two automata, we compute the intersection of one of the automata with the complement of the other. The resulting automaton is then minimized using a standard Hopcroft minimization [167].

In both the Log-based and the PL-based oracles, traces are generated from the difference automata by applying a breadth-first search until an accepting state is reached. It is worth to mention that the execution of the active learning process is deterministic.

4.5 Evaluation of Proposed Approach

In this section, we present an experiment for the evaluation of our approach. We report our study according to the guideline proposed by Runeson et al. [324].

4.5.1 Research Questions

The goal of this experiment is to evaluate whether, and to what extent, our approach can improve the efficiency of active learning in an industrial setting. We refine our goal further to the following research questions:

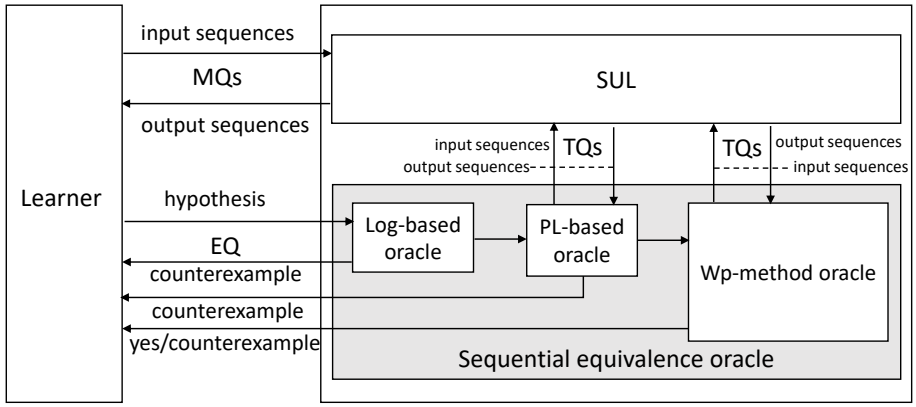


Figure 4.8: Active learning with sequential equivalence oracle

RQ1: *To what extent does our approach reduce the time for learning an SUL?* Although our approach was inspired by particular observations about far output distinction behavior, we expect our approach can reduce the time for learning any SUL, given the behavior of the SUL is deterministic and not influenced by data parameters.

RQ2: *How do the Log-based oracle and the PL-based oracle individually contribute to the time reduction together with the Wp-method?* We conjecture that both the Log-based oracle and the PL-based oracle contribute to the improvement. However, it is possible that one outperforms another, or one of them does not contribute significantly. Answering this question can help us to improve the architecture of the equivalence oracle, as well as assist in making the trade-off between efficiency and computational complexity for the approach.

RQ3: *Does combining different oracle components help?* We combine Log-based and PL-based oracles with the Wp-method oracle, as we conjecture that these two oracles contribute different behavior to the learning. However, it is possible that using only one of them with the Wp-method can already achieve the same or relatively comparable performance.

4.5.2 Component Selection

We selected cases from the 218 ASD components studied in our pilot study. As they are MDSE-based components, we know the size of the behavior of the components, and can correctly configure the Wp-method oracle. Furthermore, we can evaluate the industrial applicability of our approach.

We applied the following criteria to select the components:

1. Logs should be available. Our approach needs logs for the Log-based and PL-based oracles. Unavailability of logs makes the approach inapplicable. At ASML, software execution is logged during both normal machine production and software testing, if logging is enabled for the component.

2. The behavior of the selected components must be deterministic and not influenced by data parameters.

Limited by the availability of logs, we obtain 18 components (from 218) as study subjects. We name the components *A* to *R*; *G* is the component discussed

in the pilot study. Table 4.1 shows the number of states and input actions of these components.

Table 4.1: Features of 18 MDSE-Based Industrial Components

Component	Number of States	Number of Inputs
A	2	3
B	6	64
C	9	18
D	9	62
E	9	123
F	11	52
G	14	144
H	14	99
I	14	13
J	14	159
K	17	102
L	17	102
M	27	152
N	30	102
O	37	115
P	37	102
Q	47	103
R	80	98

4.5.3 Experiment Setup

In this section, we introduce the setup of our experiments.

4.5.3.1 Equivalence Oracle Setup

We conduct experiments with different equivalence oracle settings. We have four equivalence oracle settings in total. The equivalence oracle setting with the Wp-method alone is the control group. For the experiment groups, we have three additional equivalence oracle settings: the sequential equivalence oracles consisting of 1) Log-based, PL-based and Wp-method oracles (**EO1**), 2) Log-based and Wp-method oracles (**EO2**), 3) PL-based and Wp-method oracles (**EO3**). For each group we measure the testing time and the total active learning time. We configure a timeout of 1 hour for all experiments.

4.5.3.2 Logs and Passive Learning Results

Log-based and PL-based oracles require logs and passive learning results as inputs, respectively. The used logs are collected from the execution of unit and integration tests, containing the interactions (i.e., input and output actions) between components and their system environment. Some post-processing, such as parsing and renaming, is conducted for fitting the passive learning tools. The passive learning results are obtained using the Alergia algorithm (with a configured bound of 10) [285] provided by FlexFringe [391].

4.5.3.3 Hardware Setup

We executed all our experiments on a HP Z420 workstation, a desktop PC with an Intel Xeon E5-1620 v2, a quad core CPU consisting of cores running at 3.70

Ghz with hyperthreading, 32 gigabytes of memory, and running the Microsoft Windows 7 SP1 x64 operating system.

4.5.4 Statistical Analysis

In this section, we explain the statistical analysis used for answering RQ1-3.

4.5.4.1 RQ1

To answer RQ1 given an oracle o (EO1, EO2, EO3) we formulate the following hypotheses:

$H^o_{0_1}$: *There is no statistically significant difference between the total active learning time with the Wp-method alone and with the equivalence oracle o.*

$H^o_{a_1}$: *The total active learning time with the Wp-method alone is more than with equivalence oracle o.*

We formulate the alternative hypothesis as a directional alternative since testing the correctness of the hypothesis constructed by the learner is known to be the most expensive step in the active learning process. We expect our approach to reduce the testing time of the active learning process.

4.5.4.2 RQ2

We formulate the following hypotheses:

$H^o_{0_2}$: *There is no statistically significant difference between the total active learning time with EO2 and with EO3.*

$H^o_{a_2}$: *The total active learning time with EO3 is less than with EO2.*

The rationale behind this alternative hypothesis is that it is possible that EO3 outperforms EO2 as the passive learning results include the behavior shown in the log and potentially some other valid generalized behavior, which might further reduce the required testing time.

4.5.4.3 RQ3

Given an oracle o (EO2, EO3), the following hypotheses are formulated:

$H^o_{0_3}$: *There is no statistically significant difference between the total active learning time with EO1 and with o.*

$H^o_{a_3}$: *The total active learning time with EO1 is less than with o.*

This alternative hypothesis is a directional alternative since we expect that EO1, which is the combination of EO2 and EO3, results in shorter total learning times than when each one of the oracles is used separately.

4.5.4.4 Analysis Technique

To test the hypotheses we perform pairwise tests (RQ1: Wp-method vs. EO1, Wp-method vs. EO2, Wp-method vs. EO3; RQ2: EO3 vs. EO2; RQ3: EO1 vs. EO2, EO1 vs. EO3). Next, we adjust the p-values [48] to control the false discovery rate. Since we perform 6 pairwise tests on the same set of data, we adjust the six p-values together to control the overall Type I error rate. Finally, if the difference is observed to be statistically significant, we report the effect sizes.

4.5.5 Results

Table 4.2 presents the results of learning components A to R with different equivalence oracle settings. Using the Wp-method oracle alone, 12 out of 18

components were fully learned within 1 hour. The learning for components G, Q, K, L, R and P remained unfinished. In contrast, by applying EO1, EO2 or EO3, all components are fully learned within 1 hour. In particular, we completely learned component G, which exhibits far output distinction behavior, within 13 mins. This seems to be a promising result. Next, we further analyze the data to answer our research questions.

Table 4.2: Experiment Results (Testing Times and Total Learning Times) for 18 Industrial Components Using Wp-Method Oracle, Sequential Equivalence Oracle, Log-Based Oracle and PL-Based Oracle

Component	Wp-method		Seq. equiv. oracle (EO1)		Log-based oracle (EO2)		PL-based oracle (EO3)	
	Testing [s]	Total [s]	Testing [s]	Total [s]	Testing [s]	Total [s]	Testing [s]	Total [s]
A	0.04	0.106	0.134	0.186	0.122	0.173	0.097	0.146
B	8.676	9.328	1.230	1.820	1.120	1.704	0.814	1.407
C	2.786	3.295	1.804	3.125	2.098	3.537	1.530	4.780
D	5.813	6.671	4.615	5.326	4.286	4.965	4.570	5.268
E	319.28	322.33	131.300	134.580	130.040	132.940	111.610	115.060
F	190.46	192.57	6.565	9.685	5.782	8.456	3.554	6.414
G	timeout	timeout	778.990	788.200	816.010	826.800	765.310	772.960
H	245.65	249.42	97.083	100.450	90.290	93.340	897.460	931.330
I	27.433	29.687	8.098	10.603	9.407	11.167	61.263	90.950
J	2030.4	2039.0	1800.100	1811.000	1774.800	1787.800	1784.600	1792.700
K	timeout	timeout	3.450	7.951	3.089	7.504	4.095	8.336
L	timeout	timeout	4.231	9.781	3.772	8.957	3.834	8.593
M	1789.1	1808.7	26.357	52.127	23.578	43.365	27.089	50.341
N	221.37	229.19	6.779	16.552	6.319	15.823	7.009	16.728
O	2768.6	2783.9	719.340	735.650	718.820	733.800	718.670	734.400
P	timeout	timeout	16.528	35.635	14.817	35.234	18.456	34.503
Q	timeout	timeout	971.510	1007.600	1007.900	1038.300	984.010	1006.100
R	timeout	timeout	54.812	92.438	45.388	83.794	44.147	79.708

4.5.5.1 RQ1

To what extent does our approach reduce the time for learning a SUL? Figure 4.9 shows the total learning times for different oracles. While it clearly suggests that *overall* the Wp-method oracle is by far the slowest, this does not necessarily mean that *for each* individual component, the Wp-method oracle is slower than the other oracles. To answer this question we performed pairwise tests. Since the distribution of the learning times is skewed (cf. Figure 4.9) we opt for the pairwise Wilcoxon rank sum tests, and for Cliff's delta as the effect size measure. We interpret the Cliff's delta according to the guidelines of Cohen [83].

We observe that for all pairs of oracles $H^o_{0_1}$ can be rejected in favour of $H^o_{a_1}$ ($p \simeq 7.6 * 10^{-6}, 1.9 * 10^{-5}, 1.6 * 10^{-3}$ for EO1, EO2, EO3, respectively). Furthermore, the effect of replacing the Wp-oracle with EO1 or EO3 is medium ($\delta \simeq 0.47, 0.46$, respectively) and with EO2 it is large ($\delta \simeq 0.48$).

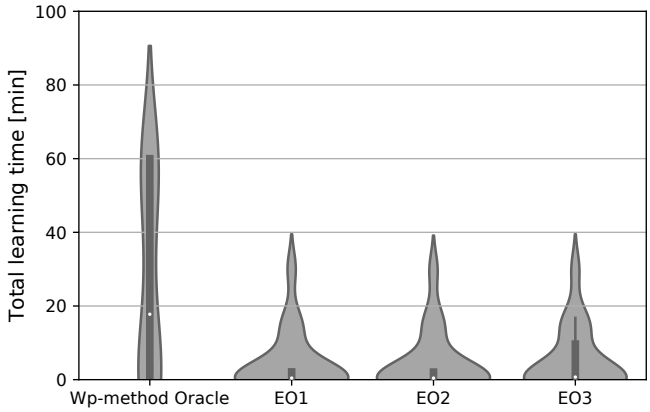


Figure 4.9: Violin plots of the total learning times with different oracles

4.5.5.2 RQ2

How do the Log-based oracle and the PL-based oracle individually contribute to the time reduction together with the Wp-method? We find that $H^o_{0_2}$ cannot be rejected ($p \simeq 0.62$).

4.5.5.3 RQ3

Does combining different oracle components help? The p-values are 0.96 and 0.93 for EO2 and EO3 respectively. Therefore, $H^o_{0_3}$ cannot be rejected.

4.5.6 Discussion

Our results show that the sequential equivalence oracle and its simplified versions, the Log-based and PL-based oracles, all significantly improve the performance of active learning. However, we could not observe a significant difference in the performance of the Log-based and PL-based oracles. The conclusion one would like to derive is that integrating log data (either in the

form of a log, or in the form of a model inferred from the log using passive learning techniques) in active learning is beneficial. However, it is not yet clear whether enhancing active learning with passive learning is always beneficial or not.

Next, we perform a closer inspection of the performance of the different oracles on individual components. To this end, we show a bar chart (Figure 4.10) of the ratios of the total learning time with other oracles with respect to the total learning time with the Wp-method alone. For the cases where learning was not finished within 1 hour with the Wp-method, we use 1 hour as the total learning time. As expected, EO1, EO2 and EO3 perform better than the Wp-method oracle alone, for most of the cases. Particularly, the total learning time for relatively large components M, K and L were significantly reduced. This suggests that our approach can potentially address the challenges of learning large systems. However, some exceptions exist. For example, for component A, the Wp-method oracle alone results in the shortest total learning time. This is because component A has only two states and three input actions (as shown in Table 4.1). In such cases, the computation required in the Log-based and PL-based oracles costs more time than sending very few TQs with the Wp-method oracle. Moreover, for components C, H and I, EO3 costs more time than the Wp-method oracle. This likely indicates the presence of a significant amount of invalid generalization in the passive learning result; therefore, extra time was spent on validating the generated traces even though no counterexample was eventually found. Based on the observations, we can expect that the performance of the PL-based oracle is influenced by the used algorithms and heuristics that generalize the logs in different ways. The completeness of behavior shown in logs (e.g., the number of traces that capture different behaviors of software) also influences the performance of both the Log-based and PL-based oracles.

An advantage of the sequential equivalence oracle is the ease with which additional components based on behavioral evidence can be integrated, further reducing the need for testing. For example, as sub-oracles, one can integrate manually crafted models or multiple models learned by different passive learning algorithms to enrich the behavioral sources. Furthermore, similarly to what we did to answer RQ2, users can analyze which sub-oracles contribute more to refining active learning, and adapt the sequential equivalence oracle to their context.

4.5.7 Threats to Validity

In this section, we discuss the limitations and threats to validity.

4.5.7.1 Limitation

The main limitation of our evaluation is that we applied our approach to 18 components for which logs are available. The unavailability of logs for a larger set of components hindered us to evaluate our approach with a better distribution in the size of components and a wider diversity of component behavior. However, as the preliminary evaluation, the promising results motivate us to evaluate our approach on a larger set of software components in the future.

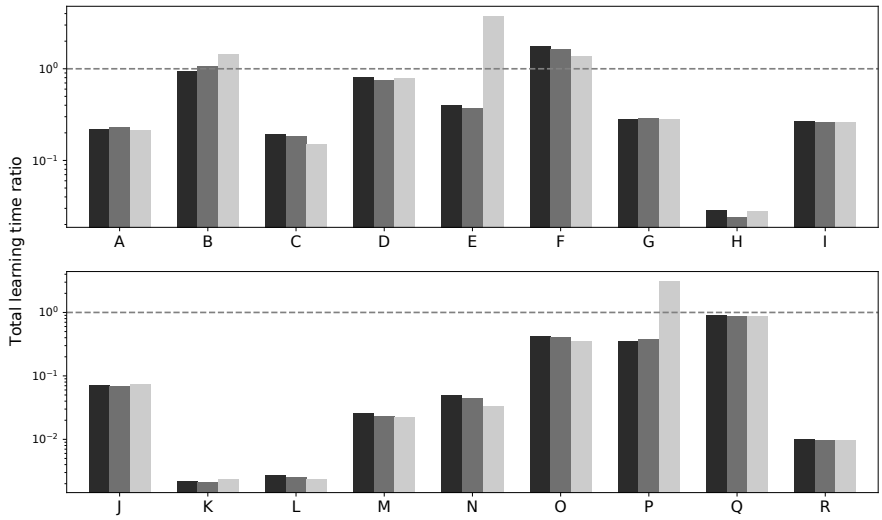


Figure 4.10: Ratios of the total learning time with 1) EO1 (black), 2) EO2 (dark grey) and 3) EO3 (light grey) with respect to the total learning time with the Wp-method alone (shown with dashed lines)

4.5.7.2 Construct Validity

This validity examines whether what we measured can quantify the efficiency of active learning. We measured the total learning time, as it intuitively measures the efficiency. However, the efficiency can possibly be quantified by such metrics as the number of MQs and TQs, since time might not be the only costly resource.

4.5.7.3 Internal Validity

In order to control the variables of our experiments, we only change the equivalence oracles in each experiment, and keep the remaining settings the same. In this study we do not consider the types of execution logs (e.g., test and production logs), the features of traces (e.g., long or short traces) in logs and the heuristics of passive learning algorithms as variables, although they can greatly influence the performance of the Log-based and PL-based oracles. Moreover, we only run the experiments once, although the time spent in different runs might slightly differ.

4.5.7.4 External Validity

This validity questions whether our conclusions are valid in a more general context. We find two threats to this validity. First, as stated, the limited number of study subjects is the main limitation of our work. We expect that the Log-based oracle can help for other systems as well because it finds counterexamples without costing any test query. Second, our study subjects are a group of components used to perform control logic of systems. A further study is required to evaluate our approach on different types of system from different companies.

4.6 Related Work

The idea of combining different techniques to solve theoretical and practical model learning problems is not new. Some hybrid learning techniques aim at enabling learning on a larger set of systems. Walkinshaw et al. [395] introduced a way to combine data mining techniques with a passive learning algorithm to learn the behavior that is influenced by data parameters. For the same purpose, Howar et al. [168] opened the black-box of the SUL by applying symbolic and static analysis techniques to iteratively refine the active learning result. Different from these approaches, our work combines techniques to improve the efficiency while guaranteeing a certain minimum behavioral coverage for deterministic and non-parameterized systems.

Smetsers et al. [347] combined conformance testing with mutation-based fuzzing methods, to enhance the equivalence oracle in active learning. This work uses a fuzzer to mutate the program tests, monitors the code coverage of the generated tests, and uses the mutated tests as a source of counterexamples. The authors experimentally showed that the hybrid approach can discover states that were not learned by using conformance testing alone. However, making use of the full potential of this approach requires the source code to be available; a 2.5 times efficiency degradation was observed when the source code was not available. Differently, our approach still treats the SUL as a black-box and therefore can be applied independently of the availability of the source code and programming language in which it was written.

To improve the quality of the models resulting from active learning, Smetsers et al. [348] used a metric to measure the distance between hypotheses and the behavior of the SUL. The proposed approach promises that the measured distance does not increase, i.e., the behavioral coverage does not decrease over time. This means that when users stop the learning, the current hypothesis is the best model the active learner has ever constructed, in terms of behavioral coverage. Our approach makes a different promise about the behavioral coverage, i.e., the learned models at least cover the execution logs.

Previous work has also shown that combining different techniques can reduce the need for testing hypothesized models with respect to SUL in active learning. Howar et al. [168] adopt partial order reduction to reduce the number of required sequences in testing. This work relies on static analysis to determine mutually independent functions (input actions), and only a single order is constructed for these functions. Instead of using white-box techniques such as partial order reduction, our approach reduces the need for testing by searching counterexamples from the execution logs and the generalization of passive learning results. As stated, finding counterexamples faster is the key to reducing the number of tests. Smeenk et al. [346] used manually crafted counterexamples to reduce the testing time. However, constructing counterexamples manually requires domain knowledge, which is not necessarily available. We use logs and passive learning results instead, and do not rely on domain knowledge.

Several approaches use logs to refine learning. Smetsers et al. [60] claimed their approach ensures that learned models cover the behavior of the logs, yet it is not clear *how* logs were integrated into the learning framework. Their work also suffers from limited evaluation and the use of artificial logs. Differently, we explicitly integrated the logs as an equivalence oracle and evaluated our

approach on a larger scale in an industrial setting. Bertolino et al. [49] proposed to build up a framework for updating the hypothesis while a networked system is evolving. This framework integrates a continuously running monitor that collects system traces at runtime, examines the mismatch between the hypothesis and the target system, and then refines the learning. However, this continuously running mechanism can be very expensive in terms of time, memory resources and infrastructure cost, thus making it less applicable. Our approach integrates pre-collected logs into the learning process. In addition, our approach enables combining logs from different sources (e.g., test execution and production), which can potentially enrich the behavioral coverage, as different logs might represent the execution of completely different use cases of the systems.

In this study, we demostrate the hybrid learning technique with the controller of embedded production systems. Since the original active learning tecchniques have been applied to various types of software systems (e.g., telecommunication systems), we believe this enhanced technique can be applied to other types of systems as well. However, as discussed by Vaandrager *et al.* [383], the family of active learning techniques is still limited to a certain class of software systems due to some foundamental challenges (e.g., difficuties to learn behavior with data operations). To address this problem, some progress has been made by Isberner [175] to infer register automata with which a stack with a finite capacity storing values from an infinite domain can be learned.

4.7 Conclusion and Future Work

In this Chapter, we started with a pilot study that evaluates the performance of a state-of-the-art active learning setup on a collection of 218 MDSE-based components provided by ASML. We observed that the active learning converged for 112 components in one hour or less. For these components, we have observed that as the total learning time increases, active learning becomes dominated by the testing phase (as shown in Section 4.3.2). Active learning did not converge for 106 models. By inspecting one of these components, we observed that the lack of convergence can be attributed to the presence of far output distinction behavior in the SUL. As far output distinction behavior is part of the regular system behavior, we expect to observe it in the execution logs.

To improve the efficiency of active learning we have proposed the sequential equivalence oracle integrating information from the execution logs and passive learning results. The sequential equivalence oracle has been evaluated on 18 industrial components. The results show that our approach can significantly reduce the total active learning time. Evaluation of the individual oracles suggests that using the Log-based oracle with the Wp-method might be sufficient to achieve good efficiency. However, considering other variables, such as the completeness of the logs and the level of generalization introduced by different passive learning algorithms, we suggest conducting a more comprehensive experiment that takes all these factors into account.

We suggest several directions for future work. First, it is valuable to conduct a more comprehensive study with the sequential equivalence oracle on a richer set of components (i.e., more components from different types of systems).

By investigating more variables, researchers can provide users a guideline to adapt the techniques in their own context. An example can be seen in Aslam's work [31] which extended the experiment with 208 model-based components from ASML using several different testing algorithms. The work has not only validated the effectiveness of the sequential equivalence oracle presented in this study, but also provided insights on the performance of different testing algorithms for the equivalence oracle of active learning. More research work is needed to study what features of traces can complement the Wp-method better, and which passive learning algorithms work best for our PL-based oracle. Researchers are suggested to investigate state-of-the-art passive learning techniques based on machine learning [188, 210]. The high precision of the learned models from these techniques can potentially help active learning infer valid behavior faster. Second, while in this study we apply the techniques to MDSE-based software, it is important to apply active learning to legacy software. Moreover, the current approach is limited to the class of systems where values of data parameters do not influence the behavior. Learning data-dependent behavior is still a challenge for model inference techniques in terms of scalability [330]. It might require researchers to open the black-box of the SUL (cf. Howar et al. [168]) and integrate static analysis techniques. Third, although we have demonstrated that the hybrid technique outperforms the state-of-the-art active learning algorithm, we suggest researchers to conduct benchmark studies to compare the hybrid technique against various kinds of model inference techniques. Existing datasets with ground truth models constructed from industrial software [178] and open-source software [87, 307] can be leveraged to conduct the comparison.

5

Modeling Practice: Why Developers Violate Guidelines

Models, as the main artifact in model-driven engineering, have been extensively used in the area of embedded domain for code generation and verification. Chapter 4 introduced a hybrid technique that infers state machines from existing codebase. We discussed the challenges that impede the practical application of active learning techniques. Due to these fundamental and practical challenges, the industry currently still relies on manual model creation by developers. In this chapter, we present our study about modeling practice (the RQ5 highlighted in Figure 5.1).

Many state machine modeling guidelines recommend that a state machine should have more than one state in order to be meaningful. Single-state state machines (SSSMs) that violate this recommendation, however, have been used in modeling cases reported in the literature. We aim for understanding the phenomenon of using SSSMs in practice, as understanding why developers violate the modeling guidelines is the first step towards improvement of modeling tools and practice.

We present an exploratory study that investigated the prevalence and role of SSSMs in the embedded domain, the reasons why developers use them, the advantages and disadvantages that developers perceive, as well as when SSSMs have been introduced to the systems. We present the result obtained from a repository mining study of 1500 state machines from 26 components at ASML, and interviews with developers.

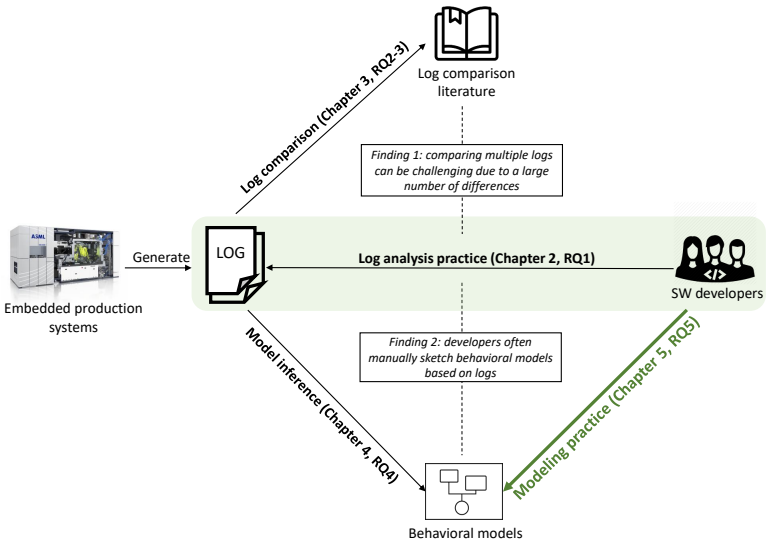


Figure 5.1: Research overview (RQ5)

5.1 Introduction

Models play a central role in model-driven software engineering (MDSE) [407]. While models are typically used to facilitate team communication and serve as implementation blueprints, in the area of embedded systems modeling, models have been widely used for such goals as code generation, simulation, timing analysis and verification [228]. One of the most popular modeling techniques used to specify the behavior of software are *state machines*.

Many guidelines have been proposed on how one should model system behavior using state machines [20, 95, 309, 331]. One of the recommendations commonly repeated both in books [20, 95, 329] is that a state machine model is only meaningful if it contains more than one state, and if each state represents different behavior. The intuition behind this guideline is that a model should contain non-trivial information, otherwise it merely clutters the presentation of ideas [20]. Single-state state machines (SSSMs)—affectionately known as "flowers" due to their graphical representation shown in Figure 5.2—violate this recommendation. From the growing body of software engineering literature, we know that software developers do not always follow recommendations or best practices and often have valid reasons not to do so [69, 291, 382].

We believe that understanding why a widespread recommendation is not followed in practice is the first step towards improvement of modeling tools and practices. In this work, we conduct an exploratory case study at ASML, the leading manufacturer of lithography machines to gain a deeper understanding of modeling practices. We employ the sequential explanatory strategy [105] which applies qualitative and quantitative analyses sequentially to explain a phenomenon. We first mine the archive for 26 components totaling 1500 models to understand the *prevalence of SSSMs* (RQ5.1) as well as *the role played by SSSMs* (RQ5.2). Then we discuss our quantitative findings with software architects to

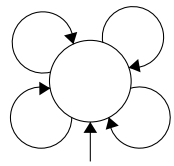


Figure 5.2: A flower model (SSSM). The circle represents the single state, and the arrows going from and to the same state represent the transitions. The incoming arrow indicates the initial transition into this state.

understand *why they opt for SSSMs* (RQ5.3), *what advantages and disadvantages of SSSMs they perceive* (RQ5.4). We then further study the question of *when these SSSMs were introduced to the system* (RQ5.5). Answering this question can help us better understand the challenges developers are facing. For example, do developers introduce SSSMs in the early phase of development or in the maintenance phase of development?

We observe that SSSMs make up 25.3% of the models considered. These SSSMs are often used with other models as *design patterns* to achieve developers' goals. We identified five such design patterns that are repeatedly used in multiple components. The used SSSMs and design patterns provided industrial evidence on how developers deal with the existing code base and tool limitations, which are the common problems in MDE adoption [228]. Given ASML has a large portion of its code base developed with the traditional software engineering practices, 20.3% of SSSMs are used on the boundary of "model world" to interface model-based components with the *existing code-based components*. Most SSSMs (64.7%) are used to circumvent the *limitations* of the modeling tools used by ASML. Apart from dealing with the common MDSE challenges, around 7.6% of SSSMs are designed to ease long-term *maintenance* of the models. Our interviews also reveal that SSSMs, as the extreme cases that remove all state information away, can pass verification easily, which is considered as both an advantage and a disadvantage by developers. It is a great challenge to design models with sufficient amount of behavioral information so that not only development but also maintenance and verification can be eased. This implies the trade-off between the effort spent on designing a model that maximizes the advantage of verification and the extra cost caused by downstream problems due to inadequate verification.

We mine the historical data of the largest state-machine-based component in the company and manually inspecting the modifications developers made during the evolution of SSSMs. We observe that the SSSMs introduced to ease maintenance and verification appeared in the early phase of component development, and their number did not increase over the years. However, over the years, more and more SSSMs are needed to deal with tool limitations, and it has become the main reason why developers introduce additional SSSMs in recent years. This observation suggests that practitioners should thoroughly evaluate the strengths and limitations of modeling tools, taking the future development of their applications into account.

Based on our results from our study, we formulate some implications for developers who would like to adopt state-machine-based solutions, as well as for tool builders and researchers.

The remainder of this chapter is organized as follows. Section 5.2 presents the preliminaries related to this study. In Section 5.3, we present our study context. In Section 5.4, we present the research method used in this study. In sections 5.5, 5.6, 5.7 and 5.8, we present our study aimed at understanding the prevalence of SSSMs, the role played by them, why developers used them, the advantage and disadvantage perceived by developers, and when SSSMs were introduced to the system. We discuss threats to validity in Section 5.9. We then discuss the implications in Section 5.10. The related work is discussed in Section 5.11. Finally, the conclusions are presented in Section 5.12.

5.2 Preliminaries

We introduce the notion of SSSM and the relevant parts of the tool-chain used at ASML.

5.2.1 Single-state State Machine

Intuitively, in its simplest form, a state machine is a collection of states and transitions between them. Some state machine modeling languages, such as UML state machines, have additional mechanisms (e.g., nested states and state variables) that can represent state information. We exclude the nested states and state variables from consideration, as the nested states and the values of state variables can be flattened into simple states [190, 298].

In our study, we consider a state machine as a single-state state machine (SSSM) if the state machine has syntactically only one state. It is a state machine that accepts all its inputs, indefinitely, and in any order. We call any other state machine a multi-state state machine (MSSM). For example, an MSSM can have more than one state, nested states or make use of state variables.

5.2.2 A State Machine Modeling Tool: ASD

Analytical Software Design (ASD) is a commercial state machine modeling tool developed by company Verum [390]. It provides users with means of designing and verifying the behavior of state machines, and subsequently generating code from the verified state machines.

5.2.2.1 Model Type and Relation

There are two types of components in a system developed with ASD, namely ASD components and foreign components. The ASD components depend on each other in a *Client-Server* manner, where a client component uses its server components to perform certain tasks. The ASD components consist of *Interface Models* (IM) and *Design Models* (DM). Each IM and DM contains a state machine. The IM specifies the external behavior of a component. It prescribes the client components of the ASD component in which order the events can be called and what replies they can expect, i.e., interface protocol. The DM implements the internal behavior of a component, specifying how it *uses* its server components. The relation *uses* is realized by three types of events: call

event, reply event and notification event (Figure 5.3, left). According to the ASD manual, an event is analogous to a method or callback that a component exposes. The declaration of a call event contains the event name, parameters and the return type. A call event with a "void" return type has "VoidReply" reply event, while the one with a "valued" return type can use all user-defined reply events. For instance, call event *task([in]p1:string, [out]p2:int):void* is a void type call event with an input and an output parameter. Notification events with output parameters are used to inform clients in synchronous or asynchronous ways, similar to callback functions in such programming languages as C and Python. An IM can be implemented by multiple DMs. In cases such as component reuse, ASD components interact with *foreign components*, non-model components implemented as handwritten code. To support communication between ASD components and foreign components, the external behavior of a foreign component is represented by an IM. Figure 5.3 (right) shows an ASD-based alarm module where ASD component *Alarm* uses ASD component *Sensor* and a foreign component *Siren*. In the remainder of the chapter, we also refer to foreign components as *code-based components*.

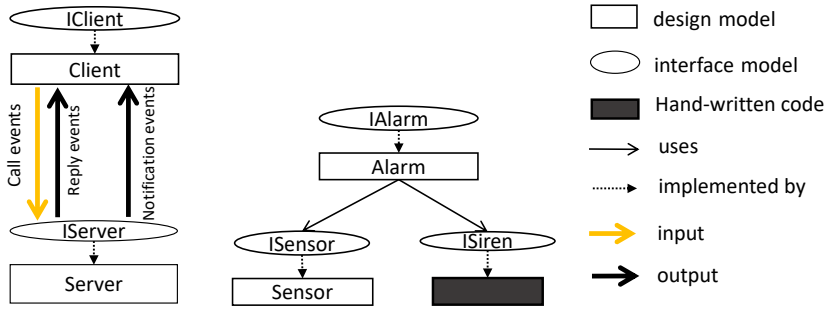


Figure 5.3: Model relations. Left: type of events. Right: example of an ASD-based system . I*** stands for an IM.

5.2.2.2 Verification and Code Generation

One of the major benefits of using ASD is the possibility to formally verify behavior of the models.

For each component, the type of verification performed by ASD can be summarized into two steps. First, ASD verifies whether each DM has correct behavior, in the sense that its behavior is deterministic and does not contain any deadlocks, or livelocks. It should also not perform illegal sequences of calls. The role of the IM in this check is just to provide the verification tool with information on which calls are considered illegal. For our alarm module example, ASD checks whether DM *Alarm* calls occur in the order specified in IMs *ISensor* and *ISiren*. Second, ASD verifies whether the DM of a component, together with the interfaces of its servers, correctly refines the IM of this component. It does this by verifying whether a formal relation, so-called Failures-Divergence Refinement relation (FDR) [109], is preserved between the DM and IM. Verifying this refinement relation guarantees that the IM can be used as an abstract representation of the DMs behavior in further analysis of

the system. For our alarm system example, ASD verifies whether DM *Alarm*, together with IMs *ISensor* and *ISiren* refines IM *IAlarm* correctly. Code in the selected target language (e.g., C++) can be automatically generated once the system is free of behavioral errors.

Note that the IM and DM have different roles, not only in system modeling, but also in the verification and code generation. The IM provides an abstract view of the behavior of a component, while DM provides a detailed view. Both IM and DM are used to understand software, communicate between engineers, and verify the behavioral correctness. However, only the DM contains the implementation details that are used to generate code.

5.3 Study Context

To get a deeper understanding of the use of SSSMs in the embedded systems industry, we conducted an exploratory case study. Case study is an empirical method aimed at investigating contemporary phenomena in a context [324, 431].

We follow the recommendation of Runeson and Höst and intentionally select a case of analysis to serve the study purpose [324]. We conduct our exploratory case study at ASML. The company uses the commercial state machine modeling tool-chain Analytical Software Design (ASD) developed by Verum [390], described in Section 5.2.2, to develop the control software of their embedded systems, providing a paradigmatic context to our study. The company uses ASD to design and verify the behavior of state machines, and subsequently generate code from the verified state machines.

We obtain all software components developed with ASD in the system, except for those that are not accessible due to international legislation or contain strategic intellectual property. These 26 software components are continuously maintained; code generated based on these models runs on the machines produced by ASML. Each software component is formed by multiple interacting IMs and DMs (i.e., multiple ASD components). In total, we obtained 924 IMs and 576 DMs, with the number of IMs per software component ranging from 2 to 349, and DMs from 0 to 284. Table 5.1 gives an overview of the 26 software components. For the sake of confidentiality, we refer to these components as A, ..., Z and cannot share the models. Note that, other than these 26, software components developed with traditional software engineering still make a large portion of the software system of the machines. Therefore, these 26 components have to interact with the existing code-based components.

5.4 Methods

We employ a sequential explanatory strategy [105] which consists of three phases, namely mining a snapshot of repositories, interviewing developers and mining historical revisions of repositories. Figure 5.4 gives a high-level overview of our research method. First, we start with a quantitative approach by mining the latest snapshot of model repositories. To answer RQ5.1, we study the prevalence of SSSMs by analyzing models of the 26 components. To answer RQ5.2, i.e., to understand the role SSSMs played, we combined two

complementary approaches. On the one hand, according to Wittgenstein [414], the meaning is determined by use. Thus, we exploit structural dependencies (cf. [25, 101]) to identify the *implemented by* and *uses* relations between IMs and DMs, i.e., the use of models. On the other hand, we expect the role of the SSSM to be reflected in its name, in the same way the names of objects have been extensively used to uncover the responsibilities of software objects [127, 196, 282].

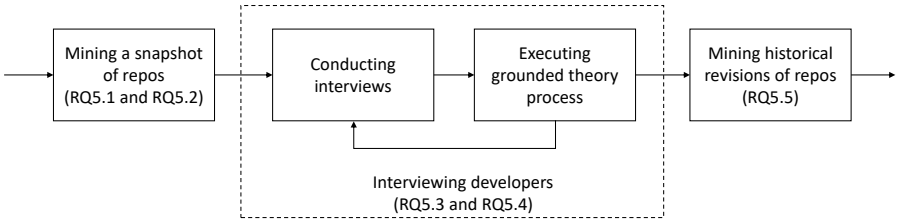


Figure 5.4: Overview of our research methods

In the qualitative phase, we conduct a series of interviews to answer RQ5.3 and RQ5.4. The interviews are recorded and audio was transcribed. To derive and refine the theory based on the obtained qualitative data, we employ Straussian grounded theory because it allows us to ask under what conditions a phenomenon occurs [358]. We opt for an iterative process to reach the saturation. It is important to note that in the sequential explanatory strategy, the results from the quantitative phase is used to inform the subsequent qualitative phase. This means the concrete study design for RQ5.3 and RQ5.4, e.g., the interview questions, is determined by the results of RQ5.1 and RQ5.2. For example, depending on the number of identified SSSMs, we opt for different interview strategies; if the number of SSSMs will be small enough, then we can request the experts to explain the reasons behind every SSSM. Otherwise, we need to prompt the discussion based on the findings we obtained from the analysis of structural dependencies and names. We detail the procedures of the qualitative phase in Section 5.7.1.

In the last phase of the study, we mine the historical revisions of models to identify when the SSSMs were introduced in the history of the model repository (RQ5.5).

5.5 Prevalence Analysis (RQ5.1)

We answer RQ5.1 by analyzing the frequency of SSSMs in the 26 components in Table 5.1.

Table 5.1: The overview of studied components, prevalence of SSSM and frequency of the identified terms for the selected state machine based projects. "-" indicates that the percentage cannot be computed because the component does not include DMs.

Component ID	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Overview																										
#IMs	19	349	22	98	15	6	22	10	10	12	29	12	29	43	3	12	16	3	41	15	49	11	2	77	3	16
#DMs	9	284	10	72	6	3	9	4	3	6	11	4	11	9	0	6	6	2	17	13	31	3	1	46	1	9
Total	28	633	32	170	21	9	31	14	13	18	40	16	40	52	3	18	22	5	58	28	80	14	3	123	4	25
Prevalence of SSSM																										
#SSSM-IMs	1	109	9	42	14	3	10	6	8	6	17	8	14	27	2	5	13	0	15	1	18	8	2	11	2	3
%SSSM-IMs	*5*	*31*	*41*	*43*	*93*	*50*	*45*	*60*	*80*	*50*	*59*	*67*	*48*	*63*	*67*	*42*	*81*	*0*	*37*	*7*	*37*	*73*	*100*	*14*	*67*	*19*
#SSSM-DMs	0	11	0	4	0	0	0	0	0	1	1	2	2	1	0	1	1	0	1	0	1	0	0	0	0	0
%SSSM-DMs	*0*	*4*	*0*	*6*	*0*	*0*	*0*	*0*	*0*	*17*	*9*	*50*	*18*	*11*	*-*	*17*	*17*	*0*	*6*	*0*	*3*	*0*	*0*	*0*	*0*	*0*
Frequency of the identified terms																										
#Exclusive	1	50	8	24	20	4	12	7	14	2	16	11	21	27	3	7	21	0	9	3	16	8	4	11	3	2
#Exclusive & Frequent	0	0	0	3	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
#Shared	0	73	3	22	2	1	2	4	4	9	8	3	6	19	0	4	3	0	12	1	13	5	0	9	1	3
#Shared & OR>1	0	45	2	14	0	0	1	0	0	5	3	1	3	5	0	1	0	0	3	1	7	1	0	8	0	3
#Shared & OR>1 & Frequent	0	14	0	3	0	0	1	0	0	0	3	0	0	4	0	0	0	0	1	0	2	1	0	2	0	0

5.5.1 Data Analysis

We analyze 1500 ASD models corresponding to components A–Z. We first convert each model into an Ecore model [354] using a tool developed by ASML. By converting ASD models into Ecore models, we can leverage EMF Model Analysis tool (EMMA) [264] which allows users to extract information from models and define metrics. In this study, we measure the number of states #*state* and the number of state variables #*sv*. The conversion from ASD models to Ecore models is lossless, i.e., the Ecore models can be converted back to the original ASD models. An SSSM is a model with #*state* $= 1$ and #*sv* $= 0$.

5.5.2 Results

Table 5.1 shows the prevalence of SSSMs in the 26 components. 25 out of 26 components contain SSSMs, making up 25.3% of the 1500 state machines. Component B is the largest component among the 26 components we consider. In component B, 31% of IMs are SSSMs while only 4% of DMs.

This tendency to use SSSMs mainly for IMs can also be observed in smaller components. In 13 out of 26 components, more than 50% of IMs are modeled as an SSSM. On the contrary, only 26 SSSM-DMs are present, and they are present in 11 out of 26 components. Furthermore, although SSSMs are generally popular among IMs, different components show different degrees of usage; SSSMs make up more than 70% of IMs in components E, I, Q, V and W while less than 10% in components A, R and T.

RQ5.1 summary:

Developers tend to use SSSMs mainly for modeling IMs. The use of SSSMs differs between components: component B has the largest portion of SSSM-IMs.

5.6 Role of SSSMs (RQ5.2)

Since SSSM-IMs are the lion's share of SSSMs, when answering RQ5.2, RQ5.3 and RQ5.4, we focus exclusively on SSSM-IMs. We start with data collection of structural relations between models and the names of models, followed by an analysis of results.

5.6.1 Data Analysis

To study what roles the SSSM-IMs play, we split IMs into three mutually exclusive locations, namely:

1. **disconnected (disc):** IMs that are neither implemented nor used by a DM.
2. **boundary (bd):** IMs that are used by at least one DM but not implemented by any DMs, or IMs that are implemented by at least one DM but not used by any DMs. They are on the boundary of "model world" independent from whether code is present on the other side of the boundary.
3. **non-boundary (nb):** IMs that are implemented by at least one DM and used by at least one DM.

Table 5.2: Number of SSSM and MSSM per location

	SSSM-IM	MSSM-IM	Total
disc	3	0	3
bd	266	195	461
nb	85	375	460
Total	354	570	924

We use EMMA [264] to extract structural relations *implemented by* and *uses* from models, and classify IMs based on these three locations.

To get complementary insights, we analyze names of models. We follow commonly used preprocessing steps (cf. [376]) including tokenization based on common naming conventions such as snake_case, camelCase and PascalCase [450], stemming [411] and removal of stop words and digits using the NLTK package [378]. We also observe that the names often contain abbreviations with the sequence of capitals, e.g., *IOStream*. Hence, prior to tokenization we manually collect a set of abbreviations from the names, compute how frequently they are used per model and remove them from the names. As a result, for each component, we obtain two document-term matrices with models acting as documents. The matrices describe the frequency of terms (including the abbreviations) that occur in a collection of the names of SSSM-IMs and MSSM-IMs, respectively.

We conjecture that the terms appearing in the SSSM-IM set while not in the MSSM-IM set (*Exclusive*), and the terms that appear in both sets (*Shared*) with high frequency in the SSSM-IM set might suggest the role of SSSM-IMs. Therefore, for each component, we further obtain the sets of *Exclusive* and *Shared* terms. To identify the "most important" shared terms, we compute the odds ratio of each term, i.e., ratio of the share of SSSM-IMs containing term t and the share of MSSM-IMs containing term t.

5.6.2 Results

Table 5.2 is a contingency table showing how many SSSM-IMs and MSSM-IMs fall into each location group. We observe that overall *bd*-models are more likely to be SSSM, while *nb*-models are more likely to be MSSM. However, such an overall assessment might obscure differences between the components, in particular since component B is much larger than the remaining components. Hence, per component, we apply statistical techniques to determine whether for an IM being an SSSM depends on the location group it belongs to. Since only component B has disconnected models, we exclude *disc* from the statistical analysis. For each component, we construct a 2×2 contingency table recording the number of SSSM-IMs and MSSM-IMs for each location. To analyze the contingency tables we opt for Fisher's exact test [114] rather than a more common χ^2 test: indeed, many components have few IMs and the normal approximation used by χ^2 requires at least five models in each group, i.e., at least 20 IMs per component. The null hypothesis of Fisher's exact test is that the type of IM (SSSM vs. MSSM) is independent of its location (*bd* vs. *nb*). Figure 5.5 shows the p-values obtained: for 9 out of 26 components the p-value is smaller

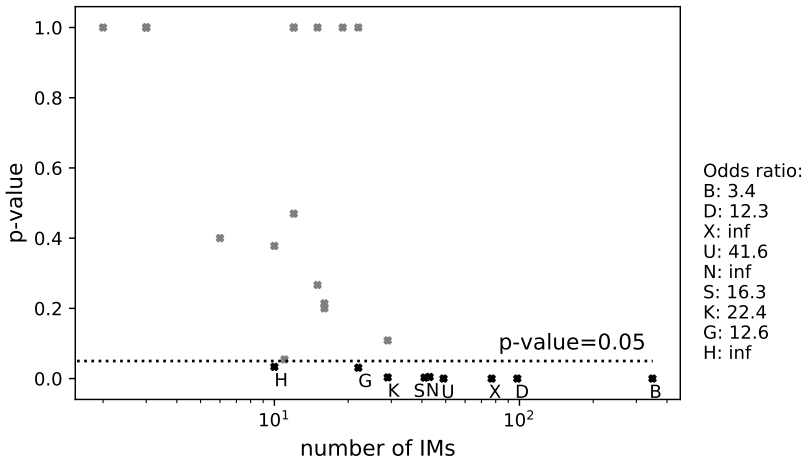


Figure 5.5: *p*-values of the Fisher's exact test vs. number of IMs

than the customary threshold of 0.05 and the odds ratio (i.e., the ratio of the share of SSSM-IMs from boundary and the share of MSSM-IMs from boundary) is larger than one. This means that we can reject the null hypothesis for these 9 components, i.e., the type of IM depends on whether it is on the boundary of the "model world". We also observe that the components where the null hypothesis can be rejected tend to have more IMs than those where the null hypothesis cannot be rejected.

Next, we identify the terms frequently used in names of the IMs. In total, we obtain 472 terms from the names of IMs for components A–Z. Table 5.1 gives an overview of the number of *Exclusive* terms, the number of *Exclusive* terms with more than five occurrences (*Exclusive*&*Frequent*), the number of *Shared* terms, and the number of *Shared* terms with an odds ratio larger than one (*Shared*&*OR>1*), as well as the number of *Shared* terms with frequencies higher than five and an odds ratio larger than one (*Shared*&*OR>1*&*Frequent*).

We observe that some terms are exclusively used in SSSM-IMs. However, only components D, K, N and S contain exclusive terms with more than five occurrences as shown in Table 5.1. The three such terms in component D are "data", "foreign" and "barrier". Components K, N and S have one such term: "access". Based on this observation, we conjecture that developers might think SSSMs particularly suit a certain functionality related to "data", "foreign", "barrier" and "access". We do not further investigate the low-frequency *Exclusive* terms because we expect them to be less likely to disclose the common roles SSSMs play.

Out of the 26 components, 22 have terms shared in SSSM-IMs and MSSM-IMs. 15 components have shared terms with an odds ratio larger than one, i.e., the models containing the term in their names are more likely to be SSSMs. As shown in Table 5.1, such terms are frequent in nine components. For component B, Figure 5.6 shows frequently occurring shared terms with an odds ratio greater than one. We anonymize the domain-specific terms and refer to them as t1,...,t5 for confidentiality reasons. Term "foreign" belongs to group

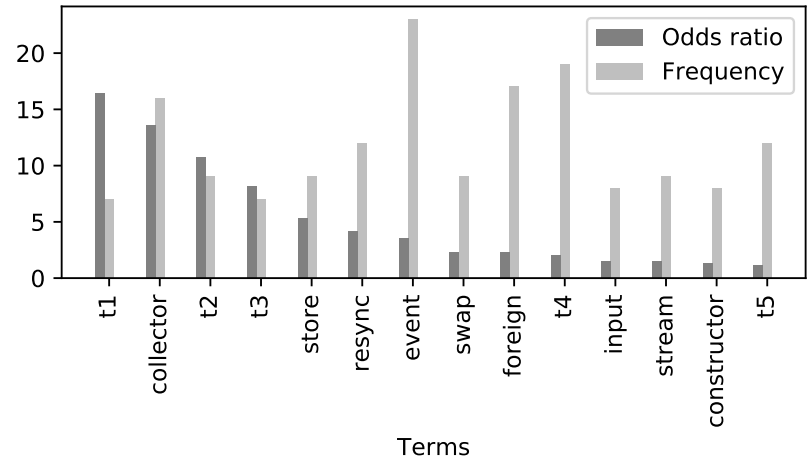


Figure 5.6: Frequency and odds ratio of terms in *Shared&OR>1&Frequent* for component B

Table 5.3: Terms from groups *Exclusive&Frequent* and *Shared&OR>1&Frequent* and the number of SSSM-IMs that contains the term

Term	collector	store	resync	event	swap	foreign	input	stream	constructor
#SSSM-IMs	16	9	12	23	9	26	8	9	8
Term	barrier	data	error	servic	access	sequenc	measur	t1,...,t14	
#SSSM-IMs	10	34	17	8	22	8	10	141	

Shared&OR>1&Frequent in component B but to group *Exclusive&Frequent* in component D. This suggests that the roles reflected by the same term might be implemented differently in different projects. Moreover, it seems that domain-specific terms are very important, as they are topping the odds-ratio list. In other eight components that have a non-empty group *Shared&OR>1&Frequent*, there are in total nine domain-specific terms identified as t6,...,t14 and five non-domain-specific terms “error”, “servic”, “seqenc”, “measur” and “data". The terms from groups *Exclusive&Frequent* and *Shared&OR>1&Frequent*, and the corresponding occurrences in the names of the SSSM-IMs from the 26 components are summarized in Table 5.3. These are the terms repeatably used in the names of SSSM-IMs.

RQ5.2 summary:

For larger components, developers use SSSMs particularly often on their boundary. Furthermore, developers repeatedly prefer terms such as “data” in the names of the SSSM-IMs.

We conjecture that terms in Table 5.3 encode the reasons why developers use SSSM-IMs and use these terms to prompt discussion in the follow-up interviews.

5.7 Interview (RQ5.3 and RQ5.4)

In this section, we present our interview methods and results for RQ5.3 and RQ5.4.

5.7.1 Procedure

Following the sequential explanatory research strategy, we refine the concrete steps for the qualitative phase based on the outcomes of the quantitative phase.

Iterative process: We start the process by considering the largest component (component B) as we expect it to produce the richest theory. We conduct semi-structured interviews with architects of the component under consideration, perform open coding of the interview transcripts to derive categories of SSSM-IMs, perform member check to mitigate the threat of misinterpretation [64], and label the SSSM-IMs in all components using the categories derived. If at this stage all SSSM-IMs have been labeled, saturation has been reached and the process terminates. Otherwise, we select a not yet considered component with the largest number of unlabeled SSSM-IMs and iterate. Figure 5.7 summarizes the process we follow.

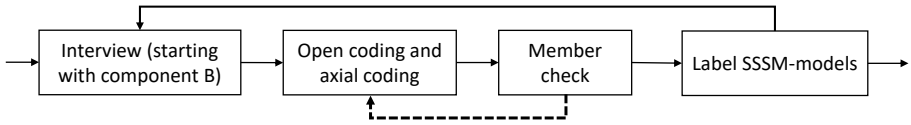


Figure 5.7: Steps in the qualitative phase

Interview design: The interview questions stem from the quantitative findings. First, reflecting on the findings for RQ5.2, we ask *why do developers use SSSMs more often on the boundary of the "model world" than in other parts?* To discuss the goals of using disconnected, boundary and non-boundary SSSM-IMs, we provide a list of SSSM-IMs for each location and ask: *what goals do you intend to achieve with an SSSM-IM in disconnected/boundary/non-boundary parts?* Next, for each term identified either as *Exclusive&Frequent* or as *Shared&OR>1&Frequent*, we provide a list of SSSM-IMs containing the term and ask questions: *what responsibilities do the term imply?*, and *why and how do you use SSSMs to implement these responsibilities?* To obtain as rich information as possible, we send a list of SSSM-IMs to our interviewees before the interviews, allowing them to refamiliarize themselves with the models. We do not disclose the interview questions prior to the interview. To answer RQ5.4, we ask developers about the *advantages of using single-state state machines and the disadvantages*. We have the interviews in a meeting room with a whiteboard. Interviewees can

draw on the whiteboard for explanation. We take photos of the whiteboard after interviews.

Coding procedures: After initial interviews, we conduct open coding on the transcripts, identifying the goals that developers attempt to achieve, the solutions that they employ and the location of the used SSSM-IMs (boundary/non-boundary/disconnected). For example, when we ask questions about term "foreign", we obtain the following answer: *"We want to create formal models, that is why we use ASD. The problem here is the outside world is not formal. So it can behave as expected or unexpected, we don't know ... If people follow the rules, all boundaries need to be armored. The important aspect is that the calls from foreign side must be accepted by every state. As foreign IM, you cannot restrict anything because you don't know the behavior of foreign (components)"*. Based on this answer, we identify the developers' goal as protecting formal models from informal and unknown foreign behavior, the solution they employ should not restrict the order of events from foreign side, and the location of the SSSM-IM is boundary.

The solution is augmented by details with photos that we took from the whiteboard. We refer to the detailed solution as a *design pattern* which are derived from multiple instances shown in our mining results and the discussions with developers about these instances. Each design pattern can be 1) an *SSSM-IM*, 2) a *combination* of an SSSM-IM and the DM(s) that implement it, or 3) a *set* of SSSM-IMs and other models. The open coding results in a set of categories that consist of *goals*, *locations* and *design patterns*. For instance, category *armoring the boundaries of models* emerges from the previous example. Next, we perform axial coding to group these categories based on the *core reason* behind, i.e., why would developers like to achieve the goal? For instance, the core reason behind *armoring the boundaries of models* is that models have to work with the existing code base. In addition, we also identify the advantages and disadvantages from our interviewees' answers.

Member check: The author of this thesis conducts the coding tasks. In order to ensure that the categories are correctly identified, we perform member check [64] with our interviewees. The member check is a validation activity that requests informant feedback to improve the accuracy of the derived the theory. This resulting adjustment on categories is represented by the dashed line in Figure 5.7.

Label SSSM-IMs: The author of this thesis reviews and labels each SSSM-IM based on the derived categories. For instance, we can determine whether a model is an instance of category *armoring the boundaries of models* by checking if it is on boundary and implements the identified design pattern.

5.7.2 Reasons of Using SSSM-IMs (RQ5.3)

We reach saturation with three face-to-face interviews and two interviews through emails. Table 5.4 provides an overview of our results. We identify four core reasons why developers use SSSM-IMs: 1) using models together with *existing code base*, 2) dealing with *tool limitations*, 3) facilitating *maintenance* and

4) easing *verification*. For each core reason, developers have at least one goal to achieve with SSSM-IMs. 353 out of 354 SSSM-IMs can be explained by the core reasons and goals listed in Table 5.4. Before discussing Table 5.4, we briefly review the model that cannot be explained by it. It is a disconnected SSSM-IM that should have been removed once it was no longer used ("dead code"). In the remainder of this section we discuss the reasons, goals and design patterns shown in Table 5.4.

5.7.2.1 Using Models Together With Existing Code Base

As mentioned, a large portion of the software base was developed with the traditional software engineering methods. Hence, the model-based components need to interact with the existing code-based components. The behavior of the models is formally verified and can only interact with each other according to the protocol specified in the IMs. By nature, when communicating with foreign components, model-based components operate under the assumption that foreign components behave as specified. However, due to the lack of formal specification, the behavior of code-based components is not formally verified and often unknown. This means that developers need a mechanism to "protect" models from non-verified and unexpected behavior of code-based components.

To achieve the goal, developers come up with design pattern D1, shown in Figure 5.8. The core idea of this pattern is to create a layer which accepts any order of calls from the code side at first, and then only forwards the allowed order of the calls to the model side. By implementing this idea, both code-based components and model-based components are not aware of the presence of each other.

Next, we discuss how the elements in the pattern work together. Developers would like to protect *Core* which is a group of models from the non-verified of code-based components *Foreign Client* and *Foreign Server*. IMs *IForeign* are SSSM-models which allow any order of input events, while DMs *Armor* forward the allowed calls specified in IMs *IProtocol* which describes the order of events expected by *Core*. In order to trace the unexpected behavior from *Foreign Client* and *Foreign Server*, DMs *Armor* also record protocol deviations with *Logger* so that it is easier to distinguish failures caused by protocol violations from failures caused by functional errors.

Table 5.4: Why developers use SSSM-IMs identified from the 26 components: the core reason, goal, location, design pattern and the number of instances (SSSM-IMs). We refer the design patterns that involve a set of models to D1,...,D5 as shown in Figure 5.8. For the sake of generalizability, we do not explain the design pattern that is used to achieve goal *EaseRefactoring* because it is specific to the semantics of the modeling language provided by ASD suite.

Core reason		**Goal**	**Location**	**Design pattern**	**#Instances**
Existing code base		*ModelArmor:* protecting verified behavior from non-verified behavior	boundary	D1	77
Tool limitations	Unable to specify data-dependent behavior	*DataEncapulation:* encapsulating data-dependent behavior into functions	boundary and non-boundary	D2	183
	Unable to select a subset of notification events	*EventCollector:* specifying individual interest for multiple clients	boundary	D3	30
	Lack of common libraries	*LibraryReuse:* reusing libraries available in general-purpose programming languages	boundary	An SSSM-IM	31
	Unable to specify global literal values	*GlobalLiteralValue:* specifying global literal values	non-boundary	Combination	2
Maintenance		*CallMapping*: reducing coupling between clients and servers	non-boundary	D4	16
		FeatureSelection: isolating product-specific features from common features	non-boundary	D5	9
		EaseRefactoring: easing event renaming	non-boundary	-	2
		Documentation: documenting events for communication within teams	disconnected	An SSSM-IM	2
Verification		*EaseVerfication:* avoiding a large state space	non-boundary	An SSSM-IM	1

5.7.2.2 Dealing With Tool Limitations

ASD suite has several limitations preventing developers from specifying the intended behavior of models. As workarounds, developers have to manually implement the behavior with general-purpose programming languages. This also results in the use of code inside model-based components. That is, a large part of the component is implemented using models while the part that cannot be implemented using models is implemented using handwritten code. This raises the need of interfacing handwritten code with models inside model-based components.

DataEncapsulation: One of the limitations of ASD suite, is the lack of a way to specify data-dependent behavior: one can declare parameters for the events in models to pass data transparently from one model to the other, but the control decision cannot be made based on a parameter value[1]. The pass-by data eventually ends up in code, where the data-dependent behavior can be programmed. To work around this limitation, developers store and manage data in handwritten code known as *data stores* inside the model-based components. The developers' goal is to have a mechanism allowing the models to read and write each piece of data. Design pattern D2 in Figure 5.8 is used to achieve the goal.

In the system under study, each piece of data in a data store is associated with an ID. For the sake of example, assume that a control decision has to be made based on the comparison of two data values associated with ID *d* persistently stored in *DataStore1* and *DataStore2* respectively. Because models can only pass data transparently, there is a need to implement handwritten code known as *Algorithm* which offers call events triggering the comparison task, and returns reply events that inform about the result. To obtain the control decision based on the comparison, DM *DataFunction* is used to fetch the data corresponding to *d* from *DataStore1* and *DataStore2*. Then it passes the fetched data to *Algorithm* to obtain the result.

Based on the received reply, *DataFunction* synchronously returns a reply to the client models that ask for a decision. For complex applications, *DataFunction* needs to intensively interact with data stores and *Algorithm* in order to derive results. To reduce the coupling between data-aware code and data-independent models, IM *im4* is an SSSM which only specifies the call events and the possible replies so that the underlying data-related interactions between code and *DataFunction* are hidden from the models that only expect a decision. Similar to IM *im4*, IM *im3* only specifies the signatures of independent functions implemented with code.

When it comes to data access, a write operation for data associated with a specific ID is required to be performed before a read operation for the corresponding data. Naturally, developers would like to specify the required order in IMs *im1* and *im2* so that the interaction protocol between *DataFunction* and these IMs is explicitly defined, and subsequently verified before code generation. However, since data-dependent behavior is not supported by ASD, *im1* and *im2* are SSSMs which only specify the signatures of call events and

[1]This limitation is intentional in order to avoid the state space explosion problem.

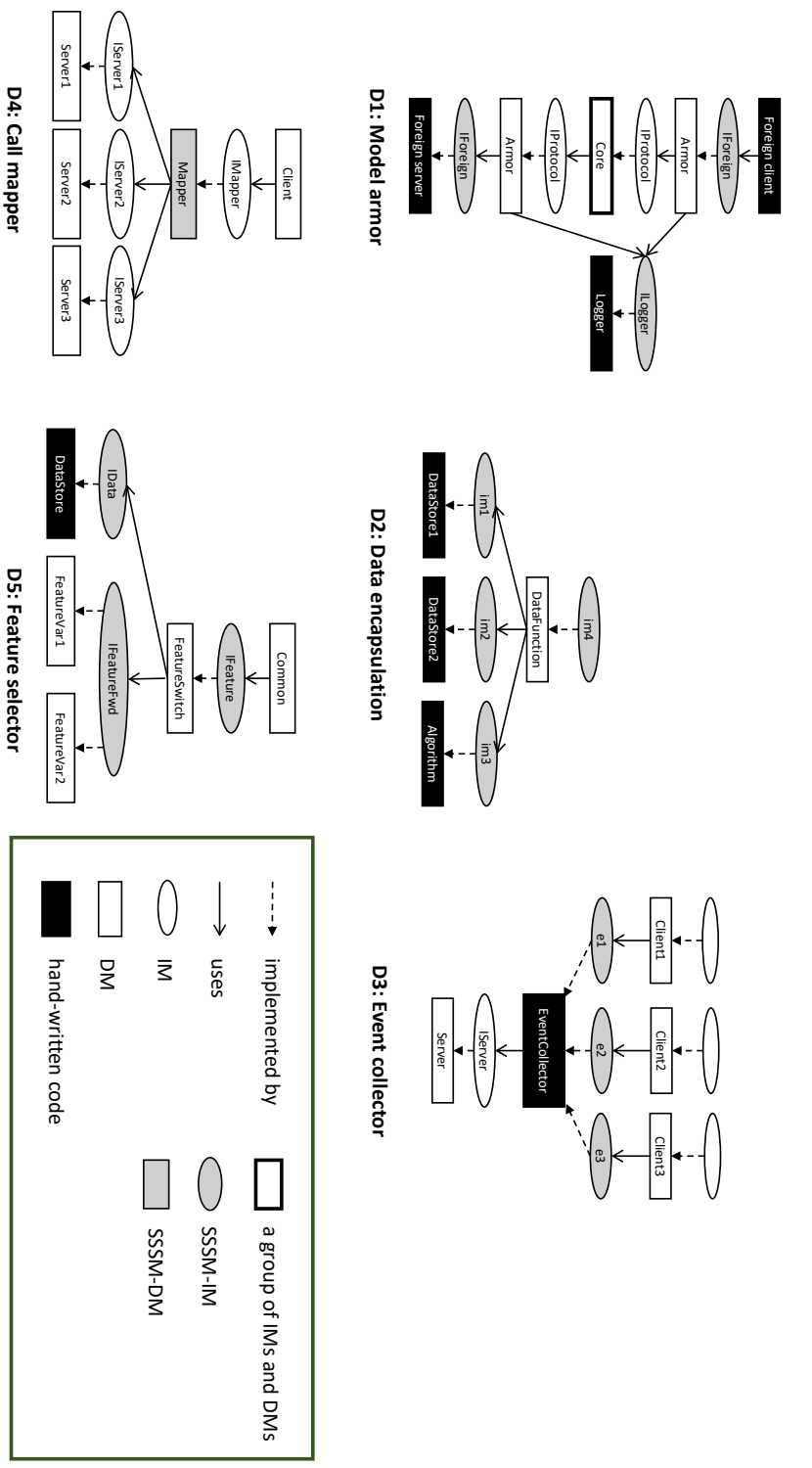


Figure 5.8: Identified design patterns D1,...,D5

replies for the intended data operations. The interaction protocol, in this case, is implicitly encoded in code for these data stores, requiring test efforts to examine correctness.

EventCollector: Another tool limitation that influences how developers design software is that client models cannot select a subset of notification events to receive from their server models. This means that the client models have to receive *all* notification events from their server models, even though some of notification events are out of their interest. To model a case where multiple client models are interested in different subsets of notification events from the same server model, design pattern D3 in Figure 5.8 is used. Instead of interfacing with the server model directly, clients interface with a handwritten *EventCollector* which works as a router forwarding each notification event to the corresponding client according to the events that developers specify with SSSM-IMs *e1*,*e2* and *e3*. Because each DM can only implement one IM developers have to inject the handwritten router between models.

LibraryReuse: ASD suite provides reusable libraries, such as a timer, implemented by models that can be used across different applications. However, the available libraries are limited compared to their counterparts available for general-purpose programming languages. For instance, one of the missing libraries is timestamp library. As a workaround, developers use handwritten code to wrap the timestamp-related operations (e.g., converting timestamp format) into functions with output parameters (e.g., for obtaining converted timestamp). The SSSM-IMs specify the signatures of the handwritten functions so that the generated code from the models can seamlessly reuse these libraries.

GlobalLiteralValue: Since ASD suite does not provide means of specifying global constants as most programming languages have, developers have to use the actual literal values wherever they need them. For example, assume that we would like to use a global constant *Size* to store the value of the buffer size set to *100*. To avoid the errors that could be introduced by hard-coding this value, developers implement SSSM-IMs and SSSM-DMs to store the value, which can be obtained by calling corresponding events. Developers specify an SSSM-IM that offers call event *getBufferSize([out]p:int):void*. In the corresponding SSSM-DM, the call is augmented with the corresponding output integer,i.e., *getBufferSize(100)*. In this case, by calling event *getBufferSize(n)*, other models that need the value can obtain variable *n* that holds integer *100*.

5.7.2.3 Facilitating Maintenance

In four cases, SSSM-IMs are used to facilitate maintenance.

CallMapping: Client models often need to call a sequence of events on different server models. To reduce the coupling between the client model and its server models, developers implement a mapper which consists of an SSSM-IM and an SSSM-DM between the client and its servers (see D4 in Figure 5.8). The SSSM-IM only specifies the signature of a void call event that can be triggered

by the client model. The mapping of the call event triggered by the client model to a sequence of intended call events on other server models is specified in the corresponding SSSM-DM.

FeatureSelection: As the system under study is specified using a principle from software product line engineering, developers separate features shared by all products from product-specific features to be configured at runtime [72]. D5 in Figure 5.8 shows a design pattern supporting this separation. For the sake of an example, assume a system needs to construct different sequences of actions for the same task based on the runtime configuration of the product type. For each product, the sequence construction is triggered by the same call event *Construct*. To hide the product-specific details from the common models, *IFeatureFwd* specifies the signature of *Construct* which is implemented by *FeatureVar1* and *FeatureVar2*. *Common*, as the common feature shared by all products, needs to call *Construct* to trigger the sequence construction on the correct variant based on the runtime configuration. However, involving *Common* in this feature selection breaks the separation of concerns, i.e., *Common* has to be aware of that different products exist. To avoid this, *FeatureSwitch* is implemented. At runtime *FeatureSwitch* reads the product type from a data store and forwards *Construct* to the appropriate product-specific implementation (i.e., *FeatureVar1* or *FeatureVar2*).

Since *IFeature* has to hide the feature selection and product-specific details from *Common*, it is identical to *IFeatureFwd* acting as an interface offering *Construct*. When *Common* calls *Construct*, the feature selection is performed, followed by the sequence construction based on the selection. *Common* is, hence, not aware of any product-specific information. Developers expect that by using this pattern, the coupling between common parts and product-specific parts can be reduced, and the variants can be extended without modifying the common parts.

EaseRefactoring: Developers also consider the ease of refactoring. Assume a model repeatedly triggers a task implemented by a sequence of *e1*,..., *e8*. Hard-coding this sequence at several invocation sites is error-prone. Moreover, any change to the sequence, such as renaming an event, has to be performed at all invocation sites. Hence, developers use a solution akin to procedure abstraction to specify a sequence of events only once and reuse it wherever needed. Since the concrete solution is specific to the semantics of ASD, we do not disclose further details.

Documentation: IMs are sometimes used to document the signatures of functions. In such cases, these SSSMs are disconnected to the rest of models in components, and used only as a way to communicate the design.

5.7.2.4 Easing Verification

Efficient verification is a key concern in modeling, as tool-chains must convert state machine specifications to a formalism before the verification step using a model checker. Verification of models with large state spaces can take significant

time, hindering the design and maintenance process. In our case study, we found a situation where an SSSM-IM is used to avoid verification on a large state space.

The intention of the developers was to create an interface such that the number of triggers on event *a* should be larger than the number of triggers on event *b*. The corresponding state space contains all possible combinations such that *a* is triggered exactly one more time than *b*, two more times, etc. During the verification step, the model checker has to visit every single state in the state space. To ease the verification step, developers simplify the model to an SSSM with events *a* and *b*, dropping the requirement that the number of triggers on event *a* should be larger than the number of triggers on event *b*: "*Scalability is a good reason to not verify this explicitly, as it does not matter if the max difference between #a - #b is 1, 2, 9 or 100. Abstracting from the exact difference makes the verification scalable, at the cost of less guaranteed correctness.*"

RQ5.3 summary:

Developers utilize SSSMs for four main reasons: 20.3% for interfacing models with *existing code*, 64.7% for overcoming *tool limitations*, 7.6% for facilitating long-term *maintenance*, and for enhancing *verification* efficiency. SSSMs are commonly used in conjunction with other models as *design patterns* to accomplish these objectives.

5.7.3 (Dis)advantages of SSSM-IMs (RQ5.4)

When it comes to the advantages and disadvantages of using SSSM-IMs, the interviewees share the same opinion. The main perceived advantage of SSSMs is the *ease* of verification. This advantage has been taken to interface with the existing code base: *"The main advantage is that a flower model is stateless, it imposes no restrictions so verification passes easily and perhaps more importantly: it is easier to implement a Foreign component faithfully"*. Indeed, there is no way to formally verify the behavior of foreign components against interface protocols. Using SSSMs allows foregin components to work with model-based components which are strictly verified.

Since SSSM-IMs impose no restrictions on the order of events, changes to the calling order on the client side also easily pass the verification, reducing the maintenance effort. However, the ease of passing verification also means that the model *"will likely always pass verification"* hiding potential bugs and compromising potential verification benefits. Taking both the advantage and the disadvantage of SSSM-IMs into account, interviewees recommend caution when using SSSM-IMs: *"people (developers) need to have a very good reason for it because it does not check anything"*. SSSMs are the extreme case where all the behavior details are abstracted away. When it comes to modeling for the purpose of verification and code generation, interviewed architects suggested that it usually takes several iterations to refine the abstraction levels of models. As suggested by these seniors, it takes several years for developers to learn *how to design models in a way that development, maintenance and verification can be facilitated.*

RQ5.4 summary:

The property of SSSMs—passing verification easily—is perceived as an advantage for easier development and maintenance of models, but also a disadvantage that might hide bugs from model checker.

5.8 When SSSMs Were Introduced to the System (RQ5.5)

In this section, we discuss data collection and analysis as well as results for RQ5.5.

5.8.1 Data Collection and Analysis

To answer RQ5.5, we examine the availability of the historical data for 26 components from Table 5.1. After an investigation, we select component B as our study subject because component B has been introduced to the system for a long time, while the other components are relatively new and have little historical data available. For example, as shared by the developers that we contacted with, component D has been deployed at the customers' machines, but it does not (yet) evolve much because there is so far no issue found by the customers and no new feature delivered.

We collect the snapshots from the Git repository of component B. The chronological order of commits from the master branch [2] is not necessarily the order of actual commits because the history of a Git repository is represented by a graph of commits rather than a linear chain of commits [54]. However, in this study, we limit our scope to the master branch due to the differences between the master branch and the other branches. First, the master branch versions the models that are ready to be reviewed by other developers while other branches version the development of machine-specific features and different releases, or the fix of certain bugs. According to the developers responsible for the components, these branches can be deleted or merged when a certain development task is finished. Second, the submitted models to the development branches may not be complete or executable (e.g., exhibiting syntactic errors). Third, developers have a different habit of committing to their own development branches (e.g., some developers commit at the end of the working day while some commit when a certain task is finished). These differences require different interpretations for the mined results. As the first step, we investigate the master branch, leaving the evolutionary differences present in other branches out of our scope.

The snapshots of the Git repository of component B are collected based on the order they appeared in the master branch. We applied the method discussed in Section 5.6.1 to identify SSSMs. For each snapshot, we measure the number of MSSM-IMs, MSSM-DMs, SSSM-IMs, SSSM-DMs as well as the number of SSSM-IMs that are used for achieving the goals we discussed in Table 5.4. By analyzing the growth of the number of these models over the years, we aim to

[2] We adhere to the terminology as used at ASML.

understand whether the trends differ between SSSMs and MSSMs, and between different SSSMs used by developers for achieving different goals.

5.8.2 Result

Figure 5.9 shows the number of MSSM-IMs, MSSM-DMs, SSSM-IMs and SSSM-DMs present in the Git repository over time. The figure shows an initial surge in 2013 because the first two Git commits are two large squashes of commits from an SVN repository which was used for the initial development of component B and has been removed after importing the latest snapshot into the Git repository.

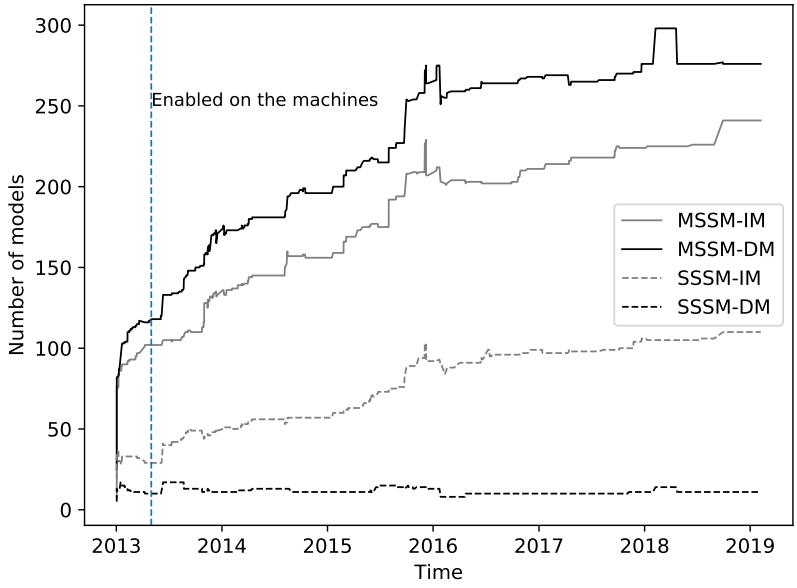


Figure 5.9: Growth of the number of models in the Git repository of component B

Overall, the total number of models in this component is growing over the years after the deployment of the component in the machines. As we learned from the developers of component B, component B is the central controller of the machines, coordinating different machine actions. Therefore, the component is likely to be extended or modified when a new feature is added to the machines. Developers started using SSSMs before the first deployment of the component and continuously introduced more SSSM-IMs over the years. The growth of all these types of models slowed down noticeably after 2016. This indicates that the component is gradually matured. In contrast, SSSM-DMs were introduced before the first deployment and their usage remains stable throughout the history.

With Figure 5.10, we zoom in on the trend for the SSSM-IMs that are used by developers for the core reasons presented in Table 5.4. After the initial development of the component, eight SSSM-IMs used for easing maintenance

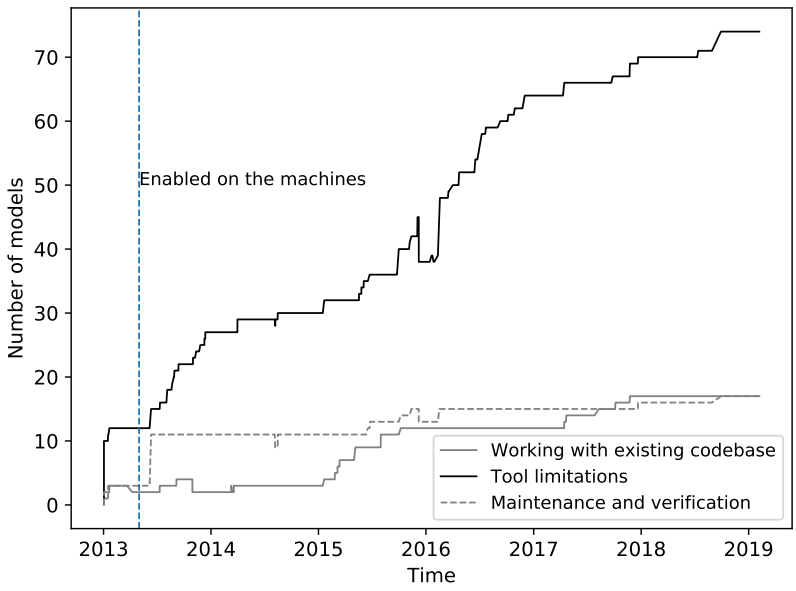


Figure 5.10: Growth of the number of SSSM-IMs used for different reasons

and verification were introduced in June 2013, and the number of the SSSM-IMs for these purposes did not grow significantly afterward. A closer look at the commit that contributes to the significant increase in June 2013 reveals that developers introduced the SSSMs when developing a machine-specific feature. These SSSMs abstract machine-specific details away from the client models (see pattern *FeatureSelection* in Figure 5.8). Differently, the number of SSSM-IMs that are used to work with the existing code base mainly increased in the period of 2015 and 2017. This implies that the need for interfacing component B with foreign components increases during the period. The number of SSSM-IMs that serves as a workaround solution to tool limitations grew continuously over the years. By further zooming in on the trends for the SSSMs used for dealing with different tool limitations as shown in Figure 5.11, we found that the demand for SSSMs for different tool limitations varies over time. The implementation of patterns *EventCollector* and *DataEncapsulation* is the main drive behind the growth. The need for the SSSMs from pattern *EventCollector* grew strikingly in 2016 and became relatively stable afterward. By inspecting the related commits, we found that the rapid growth was caused by the implementation of a system design that requires component B to subscribes to a bunch of events, receive the events during runtime, and perform the corresponding actions based on the received events. The introduced SSSMs forward the events to the target parts of component B that are responsible for the corresponding actions.

Due to another tool limitation, developers cannot specify data-dependent behavior. The SSSMs in pattern *DataEncapsulation* are used to encapsulate data-dependent behavior implemented in the foreign code. The need for data

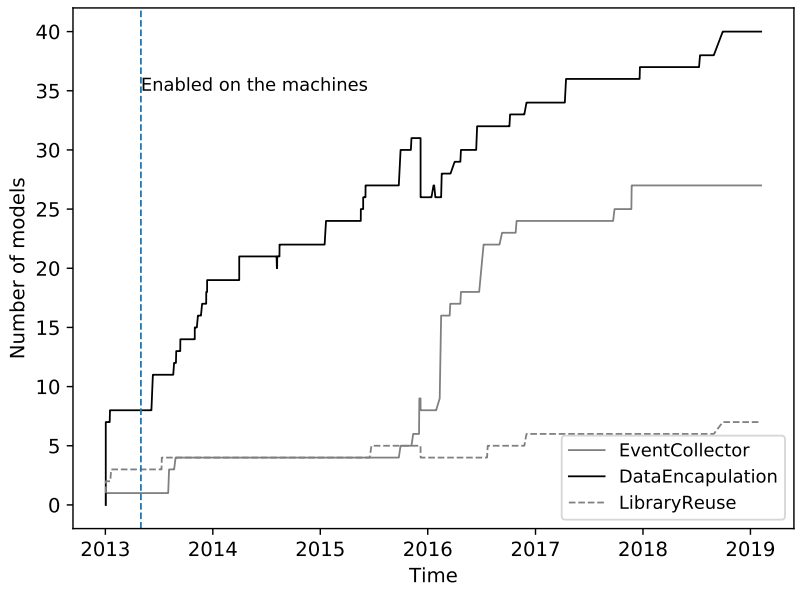


Figure 5.11: Growth of the number of SSSM-IM used to deal with different tool limitations

encapsulation with SSSMs appeared from the early phase and continuously grew as developers extend the functionalities of the component. Particularly, it became the main reason for introducing more SSSMs to the component in recent years.

RQ5.5 summary:

The SSSM-IMs used for the ease of implementing machine-specific features were introduced after the initial deployment. The SSSM-IMs used for working with the existing code base were gradually introduced as the new features were developed. The need for SSSM-IMs to deal with tool limitations continuously increases over the years. Particularly, data encapsulation is the main reason why developers introduce additional SSSMs to the component in recent years.

5.9 Threats to Validity

As any empirical study, ours is also subject to several threats of validity.

Threats to **construct validity** examine the relation between the theory and observation. Since there is no clear definition of single-state state machines in literature and guidelines, we operationalize the intuitive notion of an SSSM and provide our own definition. To ensure that our definition corresponds to the developers' perception of SSSMs, we explained our definition of SSSMs to

the interviewees and made sure that they understood it. While it is possible that some MSSMs can be reduced to SSSMs according to some formal notions of equivalence (e.g., trace equivalence), developers tend not to think about those MSSMs when talking about SSSMs. This is why we exclude this case from consideration and treat MSSMs equivalent to SSSMs as MSSMs.

Threats to **internal validity** concern factors that might have influenced the results. In our interview study, we derive our interview questions and strategy from our quantitative findings, which reduces the risk of asking meaningless questions that potentially bias our interviewees. Moreover, to avoid misinterpretation of developers' ideas, we performed member checks with our interviewees on the categories emerged from the Grounded Theory process. To assure the completeness of the reasons of using SSSMs, we conduct several iterations of interviews till all SSSMs from these 26 components can be explained by the collected reasons.

Threats to **external validity** concern the generalizability of our conclusions beyond the studied context. We studied 26 model-based components for RQ5.1 and RQ5.2. The study for RQ5.5 is limited to a single component because this component has been developed and enabled in the system for a long time, while other components are relatively new and have not evolved much since their deployment. Studying the evolution of state-machine-based software is still a challenging subject due to the lack of data. First, the use of MDSE with the purpose of verification is still very limited even though the need is already evident, as surveyed by [229]. Second, since the built-in verification tool formally verifies the correctness of models, the number of revisions developers made to these models might inherently lower than that they made to handwritten code. Lacking of data can impact the generalizability of the findings. With this preliminary study, we intend to increase the understanding of the evolutionary aspects of state-machine-based software with the evidence from industry.

Moreover, we are aware that we limited our study to the components from a single company developed with the same modeling tool. We believe the conclusions and observations derived from this context are complementary to the existing literature which mainly have broad surveys on the challenges of MDSE adoption, by providing concrete industrial examples. To increase the generalizability, one of the future directions could be replicating our study in other companies or using the models developed with other tools.

5.10 Discussion and Implication

As the main contribution, our study identified why developers use SSSM models. Firstly, we delve into the relationship between the ASD tool under study and MDSE, reflecting on the generalizability of our study and its findings (Section 5.10.1). Based on our empirical results, we then provide implications for developers (Section 5.10.2), tool builders (Section 5.10.3) and researchers (Section 5.10.4). Some of the implications derived from our empirical study are consistent with the findings provided by other surveys and interview studies on MDSE adoption. Different from these studies that provide a broad insight of MDSE adoption, our study aims for more in-depth insights into a certain phenomenon in state machine modeling, by applying mixed methods (i.e.,

interviews and repository mining) in an industry context. Therefore, we think it is still interesting to confront their conclusions with our findings.

5.10.1 ASD and MDSE

As identified by many empirical studies [75, 138], models are not only used as important artifact for designs but also primary artifact for various kinds of activities such as formal verification and code generation. The rigorous use of models is enabled by tools like ASD. Therefore, ASD represents an instance of MDSE. Moreover, ASD employs the concept of compositional verification to achieve scalability in verifying large software systems. Verification is carried out at the component level of ASD, where the relationship between a design model and its interface model, as well as the use of interface models by this design model, are examined. The premise is that if each design model (which generates code) adheres to its implemented and used interface models, all components should function correctly when combined. There are some commercial modeling tools (e.g., Cocotec[82] and Dezyne [390]) based on compositional verification. ASD is an exemplar of them.

By studying the SSSMs used in ASD modeling, we show how developers use such modeling tools. Our study also shows which part of systems are made stateless by software developers and for what purposes. The identified challenges, workaround solutions, and design patterns could be beneficial for developers and tool builders of such modeling tools. Furthermore, the derived knowledge can help researchers identify research challenges and directions to further study the practices of compositional modeling tools.

Next, we discuss these implications for developers, tool builders and researchers.

5.10.2 Implications for Developers

Consider how to integrate models with the existing code base. In our study, we found that developers introduce armoring to interface model-based components with code-based components for protecting models from unexpected behavior. In addition, we observed that the usage of SSSMs for interfacing with the existing code base is increasing as more functionalities are implemented. Our observation (in Section 5.5) suggests that practitioners should consider how to integrate models with the existing code base in a scalable way if they would like to use MDSE to develop only part of their systems that need to be integrated with handwritten code. Furthermore, practitioners may consider taking the quality (e.g., availability, scalability and maintainability) of the provided integration solutions into account when evaluating candidate modeling tools. This implication concurs with one of the challenges that has been reported to hinder MDSE adoption in companies [182, 250, 273, 353]: using MDSE together with the existing code base.

Be aware of the trade-offs that exist between using domain-specific and general-purpose programming language constructs. The trade-off between general-purpose modeling languages and domain-specific ones [388] is a frequently discussed concern about MDSE. Domain-specific languages,

on the one hand, often offer a higher degree of specialization for a certain modeling domain or purpose. On the other hand, they might be less flexible and expressive [96]. We observed a large share of SSSM-IMs are used to interface with the handwritten code for which behavior cannot be modelled with ASD because of the tool limitations (Table 5.4). Particularly, as we observed in our mining study of historical data (Section 5.8), due to the lack of means to specify data-dependent behavior with the tool, the need for encapsulating data-dependent behavior implemented with handwritten code is continually growing over the years, and has become the main reason for using SSSMs in recent years. Under-specifying the order of events for data manipulation operations require additional review and test efforts. This implies that before adopting a certain modeling language and tool, practitioners need to evaluate the benefit gained from the domain-specificity and the cost caused by the loss of general-purpose language constructs, based on their application domain, while, taking their long-term development and maintenance needs into account. This implication agrees with the suggestion provided by Corcoran [85] that "one must determine whether a given MDSE approach reduces complexity visible to the developer, or whether it simply moves complexity elsewhere in the development process."

Create reusable design using the modeling tool. Apart from developing patterns for interfacing with the existing codebase and dealing with tool limitations, we observed that developers also invest effort in creating patterns that are expected to ease long-term maintenance. They use SSSM-related design patterns to realize such software design principles as low coupling (e.g., *CallMapping*) and separation of concerns (e.g., *FeatureSelection*). Furthermore, future refactoring is facilitated with SSSMs implementing the idea of "packaging up sub-steps". We observed that these patterns were introduced in the early phase of the maintenance of component B and widely reused in other components. Our observation implies that practitioners can consider to build up reusable design patterns when using a certain modeling tool, to ease their development in future projects developed with the same tool. This implication is inline with earlier findings on MDSE adoption [169] and software engineering practice in general [22].

Balance modeling trade-off between the ease of modeling activity and the verification adequacy. As discussed by Chaudron et al. [75], developers who work with traditional UML modeling, i.e., use models merely for analysis, understanding and communication, have to make a trade-off between effort in modeling and the risk of problems caused by imperfections (e.g., incompleteness, redundancy and inconsistencies) in downstream development. For example, when a model serves as a blueprint of the protocol between two components, the under-specified parts in the model might be implemented inconsistently due to different interpretations by different developers, later incurring repair costs [75]. However, investing a lot of effort in continuously refining such blueprints is not always possible [206]. Our results imply a similar trade-off that developers need to make in the context of using models for verification. Under-specifying the behavior of models might hide defects

from the verification tools. However, spending too much effort in creating a more precise model with a restricted order of events slows down development process. Moreover, developers might need to spend more effort in performing changes on such models in their maintenance activities because passing verification becomes non-trivial.

5.10.3 Implications for Tool Builders

Help developers with integration. Our work calls for improving the support of integration of models and code-based components. The need to integrate models with the existing code base [170, 228, 408] and to integrate models from different domains [377, 379] has often been mentioned. However, not many studies propose how this integration can be facilitated by improving modeling tools. To provide suggestions to MDSE tool builders about integration, Greifenberg et al. survey eight design patterns proposed for integrating generated and handwritten object-oriented code [140]. One of the discussed design patterns is the GoF design pattern *Delegation* [125] which allows generated code (delegator) to invoke methods of the handwritten code (delegate) declared in an explicit interface (delegate interface). The *ModelArmor* design pattern we identified (Figure 5.8) implements a similar idea; DM *Armor* takes the role of delegator invoking methods of code-based components specified in IM *IForeign*. However, as opposed to *Delegation*, *ModelArmor* takes into account the different properties of models and code (i.e., verified behavior vs. non-verified and unpredictable behavior), ensuring that models are protected from the unexpected behavior of the code. Our work implies that while selecting design patterns for integration, tool builders should consider different properties of generated and handwritten code. Furthermore, tool builders can (partially) automate the implementation of the integration patterns, reducing the manual development effort.

Facilitate library reuse. Apart from interfacing with existing code-based components, we have observed that developers have to use code to implement what cannot be expressed by models (Section 5.7.2.2). For example. due to the lack of reusable common libraries, developers implement in code the behavior that requires such libraries. To address this challenge, the tool builders can work on two directions. First, one can consider enriching common functionalities often used in different applications with built-in models to reduce the needs of interfacing with libraries provided by general-purpose programming languages. Second, given rich reusable libraries in general-purpose programming languages, tools should provide a way to easily reuse these libraries, similar to the wrapping mechanism that allows, e.g., Python programs to communicate with C/C++ [45].

Meet wider specification and verification needs. We have observed that developers attempt to implement global constants with SSSMs (Section 5.7.2.2). This practice indicates the need to support concepts shared by multiple models. However, implementing such concepts is hindered by a well-known verification challenge: state explosion problem [39, 81]. Such modeling tools as Uppaal [46]

support the use of global variables (e.g., bounded integers and arrays) that can influence the control flow in the models. However, such tools have larger risk of facing state explosion when dealing with real-life applications [103]. This implies that a trade-off between supporting global variables and the risk of state explosion has to be resolved by tool designers. A possible resolution could be adopting hybrid solutions [103, 419] that translate models from one tool to another, to meet wider verification needs.

5.10.4 Implications for Researchers

As befitting an exploratory case study [324], we propose hypotheses about the use of SSSMs in modeling practice. These hypotheses should be verified in a follow-up study.

H1: *The design patterns in Section 5.7 help developers to achieve the corresponding goals*. We have seen that SSSMs are extensively used for various reasons and goals.

The studies on the effectiveness of GoF design patterns in OOP languages [125] have shown that design patterns do not always achieve the claimed advantages [21, 438]. Moreover, passing verification easily with SSSMs might be a potential risk, suggesting a need to investigate effectiveness of these SSSM-related design patterns in order to confidently apply them.

H2.1: *SSSMs shorten the development time and ease modification tasks of their client models, compared to MSSMs.* **H2.2:** *The models that use or implement SSSM-IMs have more post-release defects compared to the models that work with MSSM-IMs.* These two hypotheses are derived from our interviewees' perception (RQ4, Section 5.7.3). It is, however, unknown how SSSMs actually impact development, maintenance and verification activities. Investigating the impacts of SSSMs, the type of model that minimizes modeling effort, is a starting point toward better understanding of a trade-off between the effort spent on designing a model that maximizes the advantage of verification and the extra cost caused by downstream problems due to inadequate verification. We expect that the investigation of this trade-off can broaden the ongoing discussion of modeling trade-offs that is currently focusing on UML modeling [75, 312].

Beyond the specific hypotheses, we suggest researchers to further study the evolution of models. Given the caused permissive verification is perceived as a risk by our interviewees, we suggest proposing possible alternatives to SSSM-IMs by investigating the order in which events are *actually* being called during system operation. One can consider analyzing the execution traces of the generated code with pattern mining techniques widely studied in the field of model learning [32, 410], specification mining [219, 241] and process mining [4, 148, 406]. For example, researchers can consider applying log comparison techniques discussed in Chapter 3 to compare SSSM-IMs with models learned from execution traces. By analyzing the differences between the crafted models and actual executions, developers can refine the models with more behavioral restrictions.

In this study, we utilized an explanatory sequential design to delve into the "why" problem. However, it is important to note that this study also has an exploratory nature, as we initiate a discussion on modeling trade-offs when working with modeling tools powered by formal verification. Based on the insights gained from this research, formal approaches can be employed to analyze the level of abstraction in these models. For instance, one can investigate the gap between design models and their interface models, as well as the amount of detail exposed in interface models, in a mathematical way. Additionally, the relationship between the level of abstraction and the design of systems can also be explored. For example, it is observed that the models at the boundary are the most abstract and are intended to provide permissiveness. By combining formal and empirical approaches, one can investigate the impact of abstraction decisions on subsequent maintenance activities. Such an integrated approach can help build a deeper understanding of modeling practices from the ground up, allowing for more effective modeling and design decisions.

5.11 Related Work

In this section, we discuss several related research topics.

5.11.1 MDSE Adoption and Practice

Our study is closely related to a series of empirical studies on MDSE adoption and practice. Mohagheghi et al. [273] identified the need for more empirical evidence on MDE subjects by reviewing 25 papers. Twenty-one of these papers were experience reports from single projects, while four report comparative studies. The review study attempted to identify the benefits and limitations of MDE. As a result, the study found that the improvement of software quality, productivity gains and losses are not well-reported in these papers, making it hard to generalize the results. Therefore, the authors call for more empirical evidence on MDSE subjects to help researchers understand MDE adoption, practice, and experience. Since then, many empirical MDE studies have been conducted to understand how MDSE is being adopted and applied in practice [75, 111, 169, 170, 228, 275, 306, 407, 408]. These papers explored different dimensions of MDE adoption and practice, using mostly interviews and surveys.

Liebel et al. [228, 229] conducted a survey with 113 MDSE practitioners to assess the current state of practice and the challenges in the development of embedded systems. The study found embedded software engineers use MDSE mainly for simulation, code generation and documentation. The overall benefits gained from MDSE outweigh the negative effects of MDSE. The challenges perceived by engineers mainly lie in the sufficiency and interoperability of tools.

To understand the impact of tools on MDE adoption, Whittle et al. [408] conducted 20 interviews with MDSE practitioners, resulting in a taxonomy of tool-related considerations. In addition, the study also reveals that MDSE tools, in many cases, add complexity to the development, although it was expected to help developers deal with complexity of systems. One of the problems that contributes to the insufficiency of tools is a lack of consideration for how

developers actually work and think. To resolve this problem, there is a need to study how developers model systems and what challenges they face.

Several studies investigated challenges developers face in modeling. Pourali and Atlee [306] identified the gap between users' expectation on UML modeling tools and their actual experience. The study evaluates eight modeling tools by recruiting 18 students who are experienced with UML modeling to conduct four modeling tasks. The study found that the students mainly have difficulties in fixing inconsistencies which are most in need of consideration from tool builders. The inconsistencies and other forms of imperfection (e.g., redundancy and incompleteness) might cause downstream problems, as discussed by Chaudron et al. [75] based on a series of surveys and interviews, raising a question of how much modeling is good enough in the context of using UML as communication vehicle and implementation blueprint. Our study further reveals that this question remains when extending the use of models to verification.

Furthermore, several studies went beyond the technical aspects of MDSE adoption and practice, exploring the organizational, managerial and social factors that lead to successful adoption of MDE [169, 170, 407]. Based on a series of survey and semi-structured interviews with MDSE practitioners from industry, the authors conclude that an iterative and progressive approach, organizational commitment, and motivated users are required to successfully adopt MDSE in industry.

Similar to these studies on MDSE adoption and practice, we aimed for obtaining empirical evidence to help researchers and tool builders better understand how developers use MDSE in practice. Specifically, we enriched the existing knowledge of MDSE practice through the lens of why developers use SSSMs that is not recommended by a widespread modeling guideline, and how developers use SSSMs.

5.11.2 Guideline Adherence

Our study is inspired by the literature on how and why software developers (do not) follow programming and modeling guidelines or best practices.

A large body of literature has investigated the occurrence of violations to the common wisdom in traditional coding practice. These study observed a phenomenon that the violations often occur when the code is first introduced to the system. Tufano et al. [382] studied when and why code smells are introduced by mining software repositories. The result shows that most of the time code smells are introduced in the development phase rather than in the evolution phase that common wisdom expects, which implies that potential poor design can be detected by performing quality checks during commit activities to avoid worse problems in future. Similarly, a study on Eclipse interface usage by Eclipse third-party plug-ins found that a significant portion of Eclipse third-party plug-ins uses "bad" interfaces and the bad usage was not removed from the systems [70]. This phenomenon is further discovered by the study on how code readability changes during software evolution [300]. The result shows that unreadable code is a minority, and most of the unreadable pieces are unreadable since their creation.

The studies on guideline adherence have also been conducted to understand UML modeling practice. Lange et al. [204] formulated a collection of rules to

assess the completeness of UML models, and further explored to what extent developers violate these rules in practice. The result shows a large amount of rule violations, suggesting that the incompleteness of models should be addressed. Lange et al. [207] further conducted a controlled experiment to explore the effect of modeling conventions on defect density and modeling effort. The results show that the defect density in UML models is reduced when using modeling conventions, although the improvement is not statistically significant. Different from these studies, our study explored the *reasons* behind the violations in state-machines modeling practice.

5.11.3 Model Repository Mining

Our study is also related to the studies that mine model repositories. Pattern and clone detection is one of the goals to mine model repositories [37, 199, 355, 356]. Similar to our work, Stephan et al. [355] mine model repositories to detect patterns. The study predefined a set of patterns using models and identified the models that are similar with the patterns within a given threshold. Differently, our exploratory study identifies the patterns by mining a type of model that is not recommended by modeling guidelines and discussing the mined results with developers. As one of the main findings, we discovered several design patterns, as shown in Figure 5.8. Our study can further be extended with the pattern mining approach to detect instances of discovered patterns in the entire model base.

Some studies mined MDSE repositories to investigate the quality of handwritten code and generated code from models. He et al. [156] mined 16 MDE projects and concluded that the generated code from models present more code smells than what developers usually produce in their handwritten code. By mining MDSE repositories and non-MDSE repositories, Rahad et al. [313] further identified that handwritten code fragments from MDSE repositories suffer more from technical debt and code smells, compares to handwritten code in non-MDSE repositories. These two studies pointed out that the traditional coding guidelines are violated by code generators and developers in MDSE practice. Our study empirically shows that developers violate a widespread modeling guideline in order to integrate models with the existing code base. These studies imply that the adoption of MDSE may introduce violations to the coding and modeling guidelines that are considered to be common wisdom in software engineering practice. To improve the MDSE practice, guidelines and tools, the results of these studies call for more empirical studies to discover the workarounds and compromises that developers made when adopting MDSE.

Several studies have been conducted to mine UML models. Robles and Hebig et al. [157, 319] contributed datasets with UML diagrams mined from GitHub. The datasets enabled several mining studies to advance the understanding and techniques in UML modeling. Osman et al. [287] developed the techniques to automatically classify UML models into hand-made diagrams as part of the forward-looking development process and the diagrams reverse engineered from the source code. Raghuraman et al. [312] mined software repositories and identified that the projects with UML models present in the repositories are less prone to defects compared to projects without UML models

present in the repositories. This finding confirms the intuition that the use of UML models can improve the quality of software.

5.12 Conclusion

With the aim of understanding why developers violate a widespread modeling guideline, we conducted an exploratory study to understand under which circumstances developers use SSSMs in their practice. We first investigated the prevalence and role of SSSMs in the domain of embedded systems, as well as the reasons why developers use them and their perceived advantages and disadvantages. We employed the sequential explanatory strategy, including repository mining and interview, to study 1500 state machines from 26 components at ASML, a leading company in manufacturing lithography machines from the semiconductor industry. Then, we explored when SSSMs were introduced to the systems by mining the largest state-machine-based component from the company.

We observed that 25 out of 26 components contain SSSMs. The SSSMs make up 25.3% of the model base. Our interviews suggest that SSSMs are used to interface with the existing code, to deal with tool limitations, to facilitate maintenance and to ease verification. Our study with the historical revisions of SSSMs reveals that the need for SSSMs to deal with tool limitations grew continuously over the years.

Based on our results, we provided implications to modeling tool builders and developers. Furthermore, we formulated hypotheses about the effectiveness of SSSMs, the impacts of SSSMs on development, and maintenance and verification.

6

Conclusion

In this chapter, we first *revisit* the research questions and the main contributions presented in this thesis (Section 6.1). We then discuss the interplay between these studied topics in the context of engineering embedded production systems (Section 6.2). Next, we present the lessons learned from conducting this research (Section 6.3). Finally, we suggest future work derived from this thesis (Section 6.4).

6.1 Contributions to Research Questions

This thesis conducted empirical studies to provide insights in engineering practices for embedded production systems in industry, guided by the following research question:

> **RQ**: How do software developers engineer the software of embedded production systems?

To study this question, we conducted several empirical studies. Let us recap our research studies with Figure 6.1. The log analysis studies (Chapter 2) suggest the need for advanced log comparison tools and the preference of representing logs with behavioral models. To identify the gaps between industrial practices and the state-of-the-art techniques for log comparison, we conducted a literature review about log comparison techniques (Chapter 3). By conducting empirical studies in a real life context, we witnessed the transition that companies are experiencing to adopt MDSE. Our study about model inference techniques (Chapter 4) zoomed in on the question of how to obtain models from the existing codebase using logs, while the study about modeling practices (Chapter 5) dived into how developers use models in a hybrid system where handwritten code

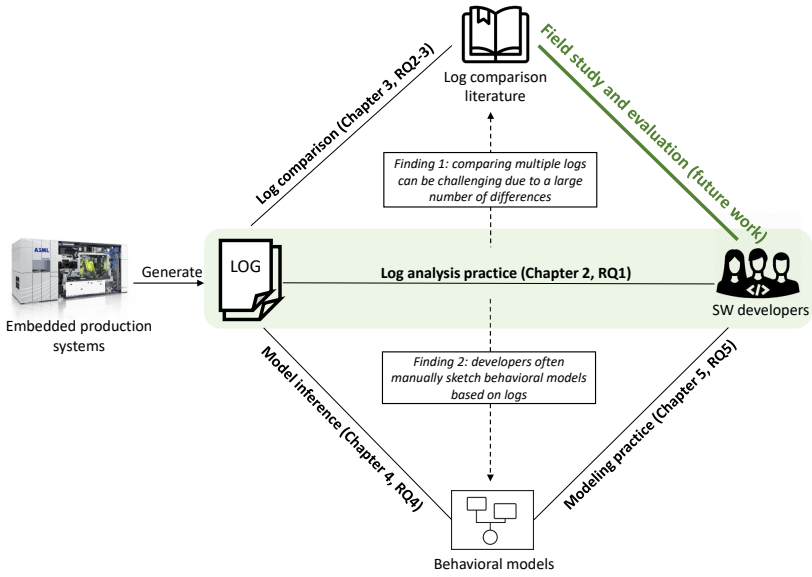


Figure 6.1: Research overview (with future work)

and generated code (from models) are co-operating with each other. These empirical studies contribute to the body of design knowledge for developing effective software analysis techniques for embedded production systems.

Next, we summarize the main contributions answering to the corresponding research questions of these studies.

Log Analysis Practice: The Exploratory Study. Logs capture the runtime behavior of systems. They are widely used as input for software analysis tools that address the complexity of software systems [439]. We are interested in how engineers use and analyze logs, and what improvements are expected from tool support. Therefore, we studied:

> **RQ1**: How do developers use logs in engineering embedded production software?

This question was addressed in Chapter 2 where an interview study with 39 developers from five companies is presented. The results of this interview study provide empirical evidence of the challenges faced by software developers for engineering embedded production systems in both log instrumentation and management activities.

We observed that the major challenge in the phase of log instrumentation lies in defining a proper logging policy which systematically formalizes logging formats, approaches and libraries. On one hand, logs are essential for engineering embedded production systems because traditional debuggers are often not applicable due to the concurrent design and critical timing

requirements of systems. On the other hand, excessive logging may greatly impact the performance of such systems. Our observations stress the challenges of software logging introduced by the critical timing properties of embedded production systems, calling for more research efforts for such systems.

As learned from the study, a suitable and well-designed logging method is the cornerstone of automatic log management and analysis for such systems. Consistent with the literature, text editors are commonly used by the interviewed developers from different companies, but cannot sufficiently provide support for developers to analyze the large amount of information shown in logs. One of the coping strategies developers adopted is to sketch behavioral models based on logs. To effectively help with analysis, developers further suggest that log analysis tools should ease the comprehension of the concurrent executions of systems, analyze and compare log information at multiple levels of abstraction, and provide actionable insights for their maintenance tasks. The insights we collected about tool support have been considered in the development of a log comparison tool by other reseachers [159] for ASML developers, demostrating the usefulness of our research outcome.

Moreover, this study also provides empirical evidence of co-evolution problems in software logging, which is not discussed in previous studies in literature. The evolution of logging code and the generated logs require the adaption of log-dependent entities (e.g., analysis tools) and tasks (e.g., interpretations of logged information) in practice. To automate the adaption, it therefore calls for more research to identify the changes made to logging code, and analyze the impact of the changes on other entities and tasks.

Log Comparison: Understanding the State of the Art. In Chapter 2, we have learned that developers use text-based tools for comparing logs and face challenges in extracting relevant information from many log differences. This is in contrast to the fact that many log comparison techniques have been proposed in the literature. As the first step towards bridging the gap, it is essential to get an overview of log comparison techniques and identify their limitations. Therefore, we ask:

RQ2: What are the existing log comparison techniques?

By studying 81 papers related to log comparison for software engineering activities, we found that issue analysis is the most frequent use case of log comparison presented in the literature, which is aligned with the use cases in practices identified in Chapter 3. This finding implies that the research efforts on log comparison techniques have targeted the major problem scenarios in industry. Interestingly, a significant number of techniques are based on behavioral model inference, performing comparison between models learned from logs, which concurs with the observation in Chapter 2 that behavioral models are favored by developers to present and comprehend log information, suggesting the potential usefulness of such model-based log comparison techniques.

Moreover, we found that the techniques presented in these papers are evaluated in a limited way. The evaluation of log comparison techniques lacks realism due to a high use of data strategies which rely on generated or simulated data and a low use of other strategies that involve human participants.

With the study presented in Chapter 2, we understand what challenges developers face in the practice of log comparison, and what tool support they expect. To apply log comparison techniques in industry, it is essential to know if the existing techniques have addressed the challenges that developers face. Therefore, we ask:

> **RQ3**: How do the existing log comparison techniques deal with the industrial challenges?

Specifically, we examined whether these techniques take event interleaving caused by concurrency into account to reduce noise, perform comparison at multiple levels of abstraction, and augment comparison results with additional information to ease inspection and comprehension. Out of 81 papers, we found that 25 of them explicitly consider at least one of these suggestions from industry and only one study considers all of them, suggesting opportunities to integrate these techniques.

The existence of multiple solutions for every challenge implies the need for empirical investigation of which solutions are most suitable for embedded production systems. For example, as we identified, in order to accurately identify event interleaving, more information e.g., thread activities, program variables, and causal relations between events, needs to be logged. The solutions based on log instrumentation introduces additional performance overhead, which might not be suitable for systems in production.

With this literature study of log comparison techniques, we conclude that more research effort is required to address the industrial challenges and evaluate the techniques thoroughly in a real development setting.

Model Inference: Combining Active and Passive Learning. Our findings presented in Chapter 2 are in line with literature that models are extensively used in the embedded domain as implementation blueprint, communication vehicles, and are the primary vehicle of model-driven software engineering (MDSE) [12]. However, to gain the potential benefit that models promise, companies are still facing the challenges to obtain models from code and logs.

Following this industrial need, we then studied techniques that can automatically infer models from code and logs. We explored the complementary

nature of active and passive learning techniques by asking the following research question:

> **RQ4**: How to combine active and passive learning techniques to infer models from code and logs?

We answered this question in Chapter 4. Although the scalability issue of active learning has been widely discussed in the literature, little work has evaluated the technique with industrial software. To bridge this gap, we first applied active learning to 218 industrial software components to show the empirical evidence that active learning indeed suffers from scalability issues due to the trade-off between learning time and completeness achieved. To tackle this issue, we proposed a hybrid technique by adding so-called sequential equivalence oracle into the original active learning framework, which allows use to utilize behavioral information from the execution logs and passive learning results. We validated this hybrid solution with 18 industrial components, and showed that the proposed solution significantly reduces the inference time and completing the behavior that is missed by the state-of-the-art technique.

This study contributes to the field of model inference by demonstrating the gain of hybrid solutions, calling for more explorations into the complementary nature of different techniques and using behavioral knowledge from other software artifacts to improve the efficiency of active learning.

Modeling Practice: Why Developers Violate Guidelines. Having learned that there is a trend in increasing the use of models for various MDSE purposes at ASML and other embedded companies [12], we further investigated the transition from traditional SE to MDSE by studying modeling practices. Different from the use of models for team communication and software design, models in MDSE are the primary artifact for further code generation, simulation, verification and validation. As a result of the transition, models are operating with the existing code base in such hybrid systems. Due to the changing role of models, the guideline formulated in a general modeling context may not necessarily be valid and sound in an MDSE context and a hybrid context (where both code and model exist). To gain insights into this hypothesis, we studied the reason behind violations of a widely accepted state machine modeling guideline, namely prohibition of single-state state machines (Chapter 5). The use of single-state state machines is an interesting phenomenon because it is contradictory to the common wisdom of using state machines to model the change of software state. We studied this phenomenon by asking:

> **RQ5**: Why do developers use single-state state machines in practice?

By mining 1500 state machines from ASML and interviewing developers to interpret our mining results, we have collected reasons for using single-state state machines (Chapter 5). Our findings show that SSSMs are widely used,

making up 25.3% of the model base. Interestingly, we found that developers have valid reasons to violate the guideline; the majority of SSSMs is used for interfacing models with the existing codebase and dealing with tool limitations. In addition, we illustrated the use of SSSM and the rationale behind the design decisions with concrete industrial examples found in the study, suggesting researchers and tool builders with possible tool improvements, and research directions. For example, we found that 8.8% of SSSMs were used for reusing libraries provided by general purpose programming languages, which suggests that tool builders should provide a mechanism to facilitate library reuse when designing modeling tools. Interestingly, we further observed that the need for dealing with tool limitation has become the main reason for using SSSMs in recent years, suggesting that practitioners should take the long-term development of projects into account while selecting and evaluating modeling tools.

The insufficiency of MDSE tools has been identified in the literature of MDSE adoption using broad surveys and interviews. Our study is complementary to the existing literature with an in-depth analysis from industry. Moreover, our study also reveals that the existing modeling theories based on traditional modeling scenario (e.g., modeling guidelines) are not necessarily applicable for various other modeling purposes.

6.2 Discussion

This thesis can be seen as a case study that investigates the engineering methodologies, processes, and tools for embedded production systems. Specifically, we presented empirical studies about logs and models in the software engineering process of such systems, identifying the challenges of log analysis and MDSE as well as the challenges companies faced in the transitional step from traditional software engineering to MDSE. In this section, we turn our attention to the interplay between the studied topics in the context of this transition.

To transit from traditional SE to MDSE, a very important step is to obtain models from which code can be generated to replace the code-based components. To achieve this replacement without altering the behavior of code-based components and changing their environment in the system, the models should precisely capture the behavior of these components. In Chapter 4, we focus on the technical challenge of inferring *exact* models from existing code-based components Our hybrid technique that combines active and passive learning mitigates the scalability problem of active learning. The effectiveness of the hybrid technique is later validated by Aslam [31] with an experiment that applies the hybrid technique in combination with various testing algorithms to 208 MDSE-based components. As discussed by Hendriks and Aslam [158], active learning still suffers from scalability challenges and requires more research efforts to improve its applicability in industry. Although researchers are advancing the model inference techniques to infer exact models in a scalable manner, companies and developers still need to heavily rely on the manual creation of models for now. In Chapter 5, we showed that the handcrafted state machine models might violate the modeling guidelines which were proposed

for traditional software modeling purposes (e.g., for blueprinting design), suggesting the need for improvement on modeling guidelines and tools.

Apart from advancing model inference techniques and improving modeling guidelines and tools, we believe one of the ways to help developers with this transition is to provide analysis techniques that can assist them in maintaining the hybrid system. Based on the studies conducted in Chapter 2 and 3, we envision that model-based log analysis techniques (i.e., analysis of models inferred from logs) can help companies and developers address some of the challenges. As discussed in Chapter 4, model inference techniques based solely on logs, by nature, result in incomplete models due to the limited observed behavior shown in logs. However, the inferred models can still be used in various software maintenance activities.

Model-based log analysis provides a unified way of comprehending and analyzing software behavior for a hybrid software system. Dealing with a hybrid software system is challenging, as it consists of software components based on different programming languages and software engineering paradigms. For example, when debugging a system which consists of model-based and code-based components, developers have to inspect different types of artifacts (i.e., code and model), get familiar with the syntax of the programming languages and apply different maintenance processes and techniques, which introduces cost in training. Inferring models from logs generated from code-based components allows developers to analyze and comprehend the system that consists of code-based and model-based software components with a unified behavioral representation, without taking the underlying differences in programming languages into account.

Subsequently, the unified behavioral representation may enable various model-based log analysis. An example that we have discussed in this thesis is model-based log comparison techniques (Chapter 3) which identify log differences at the level of models, leveraging the techniques from the field of model comparison. An extended use case of the model comparison techniques could be analyzing the differences between implemented and observed behavior to assist developers in improving and refining the handcrafted models for formal verification and code generation. By comparing the handcrafted models and the inferred models (from logs), developers can better understand the runtime interactions between components, refining the handcrafted models to avoid the problem of not restricting behavior sufficiently for formal verification (discussed in Chapter 5).

Another advantage of model-based log analysis is that it trains and prepares developers for MDSE. As shown in several empirical studies, one of the main hurdles in the adoption of MDSE is the additional cost that companies need to spend in training developers. As advised by Hutchinson et al. [169], MDSE should be introduced in a progressive manner, preparing developers with required skills. Model-based log analysis can prepare developers who are more familiar with traditional software maintenance with the use of models for various maintenance activities before introducing models as the primary artifacts.

The transition to MDSE raises the challenges of maintaining hybrid systems. In this thesis, we have collected the challenges software developers face in the

use of logs and models for embedded production systems that are undergoing a transition to MDSE, providing suggestions for researchers to improve log analysis and modeling techniques.

6.3 Lessons Learned

The studies presented in this thesis were conducted mostly in industry. In this section, we would like to summarize what we learned from conducting this research about research methodology, resource aquisition and practical challenges.

Our work contributes to the design knowledge of SE techniques and tools by gaining insights into SE practices at companies. Our work presented in Chapter 2 shows an example of collecting such design knowledge in two steps. We started an exploratory case study at a single company and later extended it with a replication study at multiple companies. We found several advantages of this approach for collecting design knowledge in industry. Conducting an exploratory study at a single company as the first step allows us to gain a deeper understanding of challenges faced by developers in a single company. The in-depth knowledge we obtained from this study helped us attract interest from other companies for our replication study. The common interest of these companies is to understand whether and to what extent their developers experience the same challenges, which is the first step towards gathering research efforts to address common SE problems. By interviewing multiple companies and synthesizing the results, we were able to discuss the common challenges and possible solutions, as well as the contextual factors that these companies featured. Furthermore, we believe that our case studies have advantages over broad surveys in terms of establishing long-term collaboration with industry. In-depth analysis of a small group of companies can help researchers prepare the internal knowledge of each company (e.g., the terms developers use and the SE process the company follows), which is useful to recruit the company for follow-up research studies.

The in-depth discussion of log analysis practices at ASML has been useful for convincing managers of ASML software development division to support our follow-up studies. As discussed in the previous section, we envision that the interplay between logs and models can be further explored and leveraged. The preference of presenting log information with models and the need for advanced log comparison tools that we observed in Chapter 2, and the category of model-based log comparison techniques that we identified in Chapter 3 leads us to the hypothesis that model-based log comparison techniques are promising solutions to help developers identify relevant information for their maintenance tasks. Therefore, we attempted to conduct a field study to evaluate a state-of-the-art model-based log comparison technique[159] for root cause analysis in industry [1]. We planned to conduct a field experiment to evaluate this technique against software developers in a more rigorous way with control

[1]Visually, this field study is represented by an arrow pointing from log comparison literature to SW developers in Figure 6.1, which would complete the diamond shape of the research overview.

and experimental groups. In this experiment design, we attempted to recruit software developers from the division where we conducted our interviews (Chapter 2), collect various resolved regression bugs and their fixes as ground truth and generate software logs from the regression and fixed software versions. We would like to ask software developers to investigate the root cause of the regression bugs using the traditional log analysis tools (i.e., text editors) and the model-based log comparison technique respectively. With this experiment setup, we intended to measure the correctness of the bug investigation and the time developers spent on the investigation. However, our attempt did not succeed. The main challenge is that the technique under study requires an integration of a log instrumentation library into the existing code base to capture communication information between software components, and the integration task requires extensive developers' domain expertises and efforts (i.e., to solve the conflicts and errors occurring in software compilations), which are hard to obtain and arrange within the given time for completing this thesis. This shortage of time and resources that hamper the ability to collect data has been widely recognized as a challenge in industry collaboration [129].

In hindsight, we would suggest researchers to propose a study plan that minimizes the resources needed from industry. In addition, it would be very helpful if the research plan is aligned with the ongoing SE activities performed by the development group in collaboration. For example, we learned that developers extensively compare execution traces (currently with text editors) to identify software bugs, when performing qualification of a new release. Planing a field study of a log comparison technique for the qualification phase of software releases could have potentially attracted more interest of software developers in participation.

6.4 Future Work

In this thesis, we studied the state-of-the-practice and state-of-the-art to understand the role and use of logs and models in practice. We suggest researchers to further investigate into identified gaps, gaining design knowledge and providing solutions to facilitate the use of logs and models in practice.

Consistent with studies conducted in other contexts [44, 146], our study in the context of the maintenance of embedded systems found that **text editors are commonly used by the interviewed companies**, despite the fact that commercial (e.g., Splunk [350]) and open-source tools (e.g., GrayLog [1]) have been made available for practitioners [154]. To further study this gap, we suggest several research directions.

Understanding the influencing factors of the adoption of log analysis tools. Tool adoption in companies is usually hindered not only by technical factors, but also non-technical factors [169, 318, 413]. To inform researchers and tool builders about the challenges companies are facing, we suggest researchers to gain a comprehensive insight into the technical and non-technical influencing factors that hinder the adoption of available log analysis tools. Moreover, it is valuable to build a corpus of evidence concerning practitioners' experience with

log analysis tools: do the adoption of available commercial and open-source tools lead to more effective log analysis?

Conducting field studies and experiments with log analysis techniques. There have been many log analysis techniques proposed in the literature. Without empirical evidence of the efficacy of these techniques, there is a danger that efforts are being wasted. As discussed by Storey et al. [359], there are multiple strategies to evaluate techniques. As we present in Chapter 3, data strategies which rely on simulated and generated data are the most used in the evaluation of log comparison techniques, which is aligned with the observation of Storey et al. about software engineering research work. In contrast, lab, respondent and field strategies which involve human participants are rarely applied. In particular, only two out of 81 studies perform field studies to evaluate their log comparison techniques. We call for field studies of log comparison techniques in a natural development setting with some controls over certain aspects of the setting. We believe that conducting field studies is an essential setp towards bridging the gap between state-of-the-art and state-of-the-pratice, as represented by the arrow shown in Figure 6.1.

By conducting our studies at ASML, **we observed the transition from traditional software engineering to MDSE and its challenges.** Several research directions can be considered to support companies to obtain and maintain models.

Leveraging various sources of behavioral information to mitigate the scalability issues in model inference. In Chapter 4, we showed that the state-of-the-art active learning technique suffers from scalability issues and demonstrated that by using behavioral information from logs, the efficiency of techniques can be greatly improved. The proposed hybrid solution provides a way to extend the original active learning framework, allowing the use of information from other sources. We suggest researchers to investigate other sources of behavioral information, such as source code, by applying static analysis techniques. Moreover, the hybrid approach is limited to the class of systems where values of data parameters do not influence the behavior. To extend model inference to a larger class of systems, learning data-dependent behavior is essential, but yet challenging in terms of scalability [330].

Refining existing modeling theories for different modeling purposes. In Chapter 5, we showed that a general modeling guideline is violated in the context of modeling for code generation and verification. Interestingly, an empirical study conducted by Rahad et al. [313] found that handwritten code fragments in hybrid systems (i.e., where generated code and handwritten code are operating with each other) suffer more from technical debt and code smells, compares to handwritten code in non-hybrid systems. Together, these studies show that the adoption of MDSE may introduce violations to the coding and modeling guidelines that are considered to be common wisdom in software engineering practice, implying the need for refining guidelines and improving MDSE tools to support this transitional step. This leads us to the

hypothesis that the existing theories (e.g., modeling guidelines and training) might require adaptions to take various modeling purposes and scenarios into account. Having seen the growing use of models as program [38], it is essential to understand to what extent the knowledge and theory obtained from traditional modeling purposes (e.g., blueprint of implementation) are applicable to other modeling purposes (e.g., code generation and verification). Answering this question can allow researchers to identify the need for different modeling purposes, and educators to prepare students with knowledge for different modeling tasks that they might perform in the future in industry.

References

[1] *A leading centralized log management solution*, `https://www.graylog.org`. Accessed: September 2022, 2020 (cited on pages 60, 175).

[2] *Acm digital library*, `https://dl.acm.org/`. Accessed: April 2021 (cited on page 69).

[3] H. van der Aa, H. Leopold, and M. Weidlich, "Partial order resolution of event logs for process conformance checking", *Decision Support Systems*, volume 136, pages 1–34, 2020. arXiv: `2007.02416` (cited on page 79).

[4] W. M. van der Aalst, *Process mining: discovery, conformance and enhancement of business processes*. Springer, 2011, volume 2 (cited on page 162).

[5] W. M. van der Aalst, *Process mining: data science in action*. Springer, 2016 (cited on page 116).

[6] W. M. van der Aalst, S. Guo, and P. Gorissen, "Comparative process mining in education: An approach based on process cubes", in *International Symposium on Data-Driven Process Discovery and Analysis*, 2013 (cited on page 79).

[7] W. M. van der Aalst, A. K. A. de Medeiros, and A. J. M. M. Weijters, "Process equivalence: Comparing two process models based on observed behavior", in *Business Process Management*, 2006 (cited on page 79).

[8] R. Accorsi and T. Stocker, "Discovering workflow changes with time-based trace clustering", in *International Symposium on Data-Driven Process Discovery and Analysis*, 2011 (cited on page 79).

[9] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: Tools for performance analysis of optimized parallel programs", *Concurrency and Computation: Practice and Experience*, volume 22, pages 685–701, 2010 (cited on page 79).

[10] V. M. Afonso, A. Kalysch, T. Müller, D. Oliveira, A. R. A. Grégio, and P. L. de Geus, "Lumus: Dynamically uncovering evasive android applications", in *Information Security Conference*, 2018 (cited on page 79).

[11] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner, "Detecting duplicate bug reports with software engineering domain knowledge", *Journal of Software: Evolution and Process*, volume 29, e1821, 2017 (cited on page 59).

[12] D. Akdur, V. Garousi, and O. Demirörs, "A survey on modeling and model-driven engineering practices in the embedded software industry", *Journal of Systems Architecture*, volume 91, pages 62–82, 2018 (cited on pages 3, 170, 171).

[13] Ü. Aksu and H. A. Reijers, "How business process benchmarks enable organizations to improve performance", *24th International Enterprise Distributed Object Computing Conference*, pages 197–208, 2020 (cited on page 79).

[14] J. P. S. Alcocer, F. Beck, and A. Bergel, "Performance evolution matrix: Visualizing performance variations along software versions", *2019 Working Conference on Software Visualization*, pages 1–11, 2019 (cited on pages 67, 79, 81, 86, 91, 92, 94, 95, 99, 102, 104).

[15] J. P. S. Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance evolution blueprint: Understanding the impact of software evolution on performance", in *First IEEE Working Conference on Software Visualization*, IEEE, 2013, pages 1–9 (cited on pages 79, 86, 94, 99, 102).

[16] J. P. S. Alcocer, A. Bergel, and M. T. Valente, "Learning from source code history to identify performance failures", *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 2016 (cited on pages 79, 86, 94, 99, 102).

[17] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Inferring hierarchical motifs from execution traces", *IEEE/ACM 40th International Conference on Software Engineering*, pages 776–787, 2018 (cited on pages 79, 84, 86, 92–96, 98, 99, 102, 104).

[18] A. K. Alves de Medeiros, W. M. van der Aalst, and A. J. Weijters, "Quantifying process equivalence based on observed behavior", *Data and Knowledge Engineering*, volume 64, pages 55–74, 2008 (cited on page 79).

[19] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz, "Using finite-state models for log differencing", in *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pages 49–59 (cited on pages 4, 11, 56, 79, 86, 92).

[20] S. W. Ambler, *The elements of UML™2.0 style*. Cambridge University Press, 2005 (cited on pages 7, 134).

[21] A. Ampatzoglou, A. Chatzigeorgiou, S. Charalampidou, and P. Avgeriou, "The effect of gof design patterns on stability: A case study", *IEEE Transactions on Software Engineering*, volume 41, pages 781–802, 2015 (cited on page 162).

[22] A. Ampatzoglou, A. Kritikos, G. Kakarontzas, and I. Stamelos, "An empirical investigation on the reusability of design patterns and software packages", *Journal of Systems and Software*, volume 84, pages 2265–2283, 2011 (cited on page 160).

[23] D. Angluin, "Learning regular sets from queries and counterexamples", *Information and computation*, volume 75, pages 87–106, 1987 (cited on pages 6, 110, 112, 114).

[24] P. O. Antonino, A. Morgenstern, and T. Kuhn, "Embedded-software architects: It's not only about the software", *IEEE Software*, volume 33, pages 56–62, 2016 (cited on page 12).

[25] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software", in *6th International Workshop on Program Comprehension*, IEEE, 1998, pages 153–160 (cited on page 139).

[26] M. Apel, C. Bockermann, and M. Meier, "Measuring similarity of malware behavior", in *IEEE 34th Conference on Local Computer Networks*, IEEE, 2009, pages 891–898 (cited on page 79).

[27] P. Ardimento, M. L. Bernardi, M. Cimitile, and F. M. Maggi, "Evaluating coding behavior in software development processes: A process mining approach", *IEEE/ACM International Conference on Software and System Processes*, pages 84–93, 2019 (cited on pages 79, 81, 86, 88, 92, 101).

[28] C. Artho, K. Havelund, and S. Honiden, "Visualization of concurrent program executions", *31st Annual International Computer Software and Applications Conference*, volume 2, pages 541–546, 2007 (cited on page 58).

[29] S. A. Asadollah, R. Inam, and H. A. Hansson, "A survey on testing for cyber physical system", in *International Conference on Testing Software and Systems*, 2015 (cited on page 15).

[30] K. Aslam, Y. Luo, R. Schiffelers, and M. van den Brand, "Refining active learning to increase behavioral coverage", in *ACM WomENcourage*, 2018 (cited on page 120).

[31] K. Aslam, "Deriving behavioral specifications of industrial software components", English, Proefschrift., Ph.D. dissertation, Mathematics and Computer Science, 2021 (cited on pages 132, 172).

[32] K. Aslam, L. Cleophas, R. Schiffelers, and M. van den Brand, "Interface protocol inference to aid understanding legacy software components", *Software and Systems Modeling*, volume 19, pages 1519–1540, 2020 (cited on pages 117, 162).

[33] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling", in *Proceedings of the 32nd ACM/IEEE international conference on Software Engineering*, 2010, pages 95–104 (cited on page 56).

[34] A. Augusto, A. Armas-Cervantes, R. Conforti, M. Dumas, M. L. Rosa, and D. Reissner, "Abstract-and-compare: A family of scalable precision measures for automated process discovery", in *International Conference on Business Process Management*, 2018 (cited on page 79).

[35] E. Avllazagaj, Z. Zhu, L. Bilge, D. Balzarotti, and T. Dumitras, "When malware changed its mind: An empirical study of variable program behaviors in the real world", in *USENIX Security Symposium*, 2021 (cited on page 79).

[36] A. Babenko, L. Mariani, and F. Pastore, "Ava: Automated interpretation of dynamically detected anomalies", in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pages 237–248 (cited on pages 79, 81, 86).

[37] O. Babur, "Clone detection for ecore metamodels using n-grams.", in *MODELSWARD*, 2018, pages 411–419 (cited on page 165).

[38] O. Badreddin, R. Khandoker, A. Forward, O. Masmali, and T. C. Lethbridge, "A decade of software design and modeling: A survey to uncover trends of the practice", in *Proceedings of the 21th ACM/IEEE international conference on model driven engineering languages and systems*, 2018, pages 245–255 (cited on page 177).

[39] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques", *ACM Computing Surveys*, volume 51, page 50, 2018 (cited on page 161).

[40] N. P. Ballambettu, M. A. Suresh, and R. J. C. Bose, "Analyzing process variants to understand differences in key performance indices", in *29th International Conference on Advanced Information Systems Engineering*, Springer, 2017, pages 298–313 (cited on page 79).

[41] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines", *Empirical Software Engineering*, volume 27, 2020 (cited on page 62).

[42] L. Bao, Q. Li, P. Lu, J. Lu, T. Ruan, and K. Zhang, "Execution anomaly detection in large-scale systems through console log analysis", *Journal of Systems and Software*, volume 143, pages 172–186, 2018 (cited on pages 79, 81, 83, 86, 94, 99, 102).

[43] L. Bao, N. Busany, D. Lo, and S. Maoz, "Statistical log differencing", *34th IEEE/ACM International Conference on Automated Software Engineering*, pages 851–862, 2019 (cited on pages 4, 11, 56, 79, 81, 86, 90, 92).

[44] T. Barik, R. DeLine, S. M. Drucker, and D. Fisher, "The bones of the system: A case study of logging and telemetry at microsoft", *IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 92–101, 2016 (cited on pages 5, 12, 14, 19, 48–51, 53, 60, 65, 99, 175).

[45] D. M. Beazley, "Swig: An easy to use tool for integrating scripting languages with c and c++.", in *Tcl/Tk Workshop*, 1996, page 43 (cited on page 161).

[46] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, *Uppaal 4.0*, 2006 (cited on page 161).

[47] A. Benbachir, I. F. D. Melo, M. R. Dagenais, and B. Adams, "Automated performance deviation detection across software versions releases", *IEEE International Conference on Software Quality, Reliability and Security*, pages 450–457, 2017 (cited on pages 79, 81, 84, 86, 89).

[48] Y. Benjamini and Y. Hochberg, "Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing", *Journal of the Royal Statistical Society. Series B (Methodological)*, volume 57, pages 289–300, 1995 (cited on page 124).

[49] A. Bertolino, A. Calabrò, M. Merten, and B.Steffen, "Never-stop learning: Continuous validation of learned models for evolving systems through monitoring", *ERCIM News*, volume 2012, 2012 (cited on page 131).

[50] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst, "Visualizing distributed system executions", *ACM Transactions on Software Engineering and Methodology*, volume 29, pages 1 –38, 2020 (cited on pages 57–59, 79, 81, 86, 87, 92, 94, 96, 97, 102, 103).

[51] F. Bezerra and J. Wainer, "Algorithms for anomaly detection of traces in logs of process aware information systems", *Information Systems*, volume 38, pages 33–44, 2013 (cited on page 79).

[52] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior", *IEEE transactions on Computers*, volume 100, pages 592–597, 1972 (cited on pages 58, 86, 110, 112).

[53] C. Bird, "Interviews", in *Perspectives on Data Science for Software Engineering*, Morgan Kaufmann, 2016 (cited on pages 15, 16).

[54] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git", in *6th IEEE International Working Conference on Mining Software Repositories*, IEEE, 2009, pages 1–10 (cited on page 154).

[55] A. Bolt, W. M. van der Aalst, and M. De Leoni, "Finding process variants in event logs: (short paper)", in *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE*, Springer, 2017, pages 45–52 (cited on page 79).

[56] A. Bolt, M. de Leoni, and W. M. van der Aalst, "A visual approach to spot statistically-significant differences in event logs based on process metrics", in *Proceedings of 28th International Conference on Advanced Information Systems Engineering*, Springer, 2016, pages 151–166 (cited on page 79).

[57] A. Bolt, M. de Leoni, and W. M. van der Aalst, "Process variant comparison: Using event logs to detect differences in behavior and business rules", *Information Systems*, volume 74, pages 53–66, 2018 (cited on page 79).

[58] M. Boltenhagen, T. Chatain, and J. Carmona, "Optimized sat encoding of conformance checking artefacts", *Computing*, volume 103, pages 29–50, 2020 (cited on page 79).

[59] M. Boltenhagen, T. Chatain, and J. Carmona, "Model-based trace variant analysis of event logs", *Information Systems*, volume 102, page 101 675, 2021 (cited on page 79).

[60] P. van den Bos, R. Smetsers, and F. Vaandrager, "Enhancing Automata Learning by Log-Based Metrics", in *International Conference on Integrated Formal Methods*, Springer, 2016, pages 295–310 (cited on page 130).

[61] R. P. J. C. Bose, A. Gupta, D. Chander, A. Ramanath, and K. Dasgupta, "Opportunities for process improvement: A cross-clientele analysis of event data using process mining", in *International Conference on Service Oriented Computing*, 2015 (cited on page 79).

[62] G. H. Broadfoot, "Asd case notes: Costs and benefits of applying formal methods to industrial control software", in *International Symposium on Formal Methods*, Springer, 2005, pages 548–551 (cited on page 15).

[63] S. vanden Broucke, J. Munoz-Gama, J. Carmona, B. Baesens, and J. Vanthienen, "Event-based real-time decomposed conformance analysis", in *On the Move to Meaningful Internet Systems: OTM 2014 Conferences: Confederated International Conferences: CoopIS, and ODBASE*, 2014 (cited on page 79).

[64] E. Buchbinder, "Beyond checking: Experiences of the validation interview", *Qualitative Social Work*, volume 10, pages 106–122, 2011 (cited on pages 17, 145, 146).

[65] M. Bugalho and A. L. Oliveira, "Inference of regular languages using state merging algorithms with search", *Pattern Recognition*, volume 38, pages 1457–1467, 2005 (cited on page 111).

[66] J. C. Buijs and H. A. Reijers, "Comparing business process variants using models and event logs", in *15th International Conference on Enterprise, Business-Process and Information Systems Modeling*, Springer, 2014, pages 154–168 (cited on page 79).

[67] A. Burattin, G. Guizzardi, F. M. Maggi, and M. Montali, "Fifty shades of green: How informative is a compliant process trace?", in *31st International Conference on Advanced Information Systems Engineering*, Springer, 2019, pages 611–626 (cited on page 79).

[68] V. Bushong, R. Sanders, J. Curtis, M. Du, T. Cerny, K. Frajtak, M. Bures, P. Tisnovsky, and D. Shin, "On matching log analysis to source code: A systematic mapping study", in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, 2020, pages 181–187 (cited on page 106).

[69] J. Businge, A. Serebrenik, and M. G. van den Brand, "Analyzing the eclipse API usage: Putting the developer in the loop", in *17th European Conference on Software Maintenance and Reengineering*, IEEE Computer Society, 2013, pages 37–46 (cited on pages 7, 134).

[70] J. Businge, A. Serebrenik, and M. G. Van Den Brand, "Eclipse api usage: The good and the bad", *Software Quality Journal*, volume 23, pages 107–141, 2015 (cited on page 164).

[71] J. Candido, M. F. Aniche, and A. van Deursen, "Contemporary software monitoring: A systematic literature review", *ArXiv*, volume abs/1912.05878, 2019 (cited on page 106).

[72] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey, "An overview of dynamic software product line architectures and techniques: Observations from research and industry", *Journal of Systems and Software*, volume 91, pages 3–23, 2014 (cited on page 152).

[73] D. Chapp, D. Rorabaugh, K. Sato, D. H. Ahn, and M. Taufer, "A three-phase workflow for general and expressive representations of nondeterminism in HPC applications", *International Journal of High Performance Computing Applications*, volume 33, pages 1175–1184, 2019 (cited on pages 79, 86).

[74] T. Chatain, M. Boltenhagen, and J. Carmona, "Anti-alignments - measuring the precision of process models and event logs", *Information Systems*, volume 98, page 101 708, 2019 (cited on page 79).

[75] M. R. V. Chaudron, W. Heijstek, and A. Nugroho, "How effective is uml modeling?", *Software & Systems Modeling*, volume 11, pages 571–580, 2012 (cited on pages 159, 160, 162–164).

[76] B. Chen and Z. M. Jiang, "Characterizing logging practices in java-based open source software projects - a replication study in apache software foundation", *Empirical Software Engineering*, volume 22, pages 330–374, 2017 (cited on pages 48, 50, 53, 54, 57, 59).

[77] B. Chen and Z. M. Jiang, "A survey of software log instrumentation", *ACM Computing Surveys*, volume 54, pages 1–34, 2021 (cited on pages 47, 54, 55, 60, 105).

[78] T. M. Chilimbi, B. Liblit, and K. Mehra, "Holmes: Effective statistical debugging via efficient path profiling", pages 1–11, 2009 (cited on pages 79, 86, 94, 98, 102).

[79] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering", in *12th International IEEE Enterprise Distributed Object Computing Conference*, IEEE, 2008, pages 222–231 (cited on page 59).

[80] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: An empirical study on software development for the cloud", *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2014 (cited on page 65).

[81] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking", in *Informatics*, Springer, 2001, pages 176–194 (cited on page 161).

[82] Cocotec, `https://cocotec.io/`. Accessed: September 2022 (cited on page 159).

[83] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988 (cited on page 127).

[84] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, H. Zheng, *et al.*, "Bandera: Extracting finite-state models from java source code", in *Proceedings of the 2000 International Conference on Software Engineering*, IEEE, 2000, pages 439–448 (cited on page 6).

[85] D. Corcoran, "The good, the bad and the ugly: Experiences with model driven development in large scale projects at ericsson", in *European Conference on Modelling Foundations and Applications*, Springer, 2010, pages 2–2 (cited on page 160).

[86] F. Corradini, C. Luciani, A. Morichetta, and A. Polini, "Process variance analysis and configuration in the public administration sector", in *International Conference on Recent Trends and Applications in Computer Science and Information Technology*, 2021 (cited on page 79).

[87] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller, "Automatically generating test cases for specification mining", *IEEE Transactions on Software Engineering*, volume 38, pages 243–257, 2011 (cited on page 132).

[88] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller, "Generating test cases for specification mining", in *International Symposium on Software Testing and Analysis*, 2010 (cited on page 11).

[89] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java", in *19th European Conference on Object-Oriented Programming*, Springer, 2005, pages 528–550 (cited on pages 79, 86, 91).

[90] D. Das, M. Schiewe, E. Brighton, M. Fuller, T. Cerny, M. Bures, K. Frajtak, D. Shin, and P. Tisnovsky, "Failure prediction by utilizing log analysis: A systematic mapping study", in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, 2020, pages 188–195 (cited on page 105).

[91] M. De Leoni, F. M. Maggi, and W. M. van der Aalst, "An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data", *Information Systems*, volume 47, pages 258–277, 2015 (cited on pages 79, 80).

[92] M. De Leoni, W. M. van der Aalst, and M. Dees, "A general process mining framework for correlating, predicting and clustering dynamic behavior based on event logs", *Information Systems*, volume 56, pages 235–257, 2016 (cited on page 79).

[93] J. De Weerdt, S. Vanden Broucke, J. Vanthienen, and B. Baesens, "Active trace clustering for improved process discovery", *IEEE Transactions on Knowledge and Data Engineering*, volume 25, pages 2708–2720, 2013 (cited on page 79).

[94] C. Deknop, J. Fabry, K. Mens, and V. Zaytsev, "Improving a software modernisation process by differencing migration logs", in *21st International Conference on Product-Focused Software Process Improvement*, Springer, 2020, pages 270–286 (cited on pages 79, 81).

[95] A. Dennis, B. H. Wixom, and D. Tegarden, *Systems Analysis and Design UML Version 2.0*. Wiley, 2009 (cited on page 134).

[96] A. Deursenvan Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography", *ACM Sigplan Notices*, volume 35, pages 26–36, 2000 (cited on page 160).

[97] W. Dickinson, D. Leon, and A Fodgurski, "Finding failures by cluster analysis of execution profiles", in *Proceedings of the 23rd International Conference on Software Engineering*, IEEE, 2001, pages 339–348 (cited on pages 79, 86).

[98] M. Diep, S. Elbaum, and M. Dwyer, "Trace normalization", in *19th International Symposium on Software Reliability Engineering*, IEEE, 2008, pages 67–76 (cited on pages 79, 81, 86, 94, 96, 97, 102, 103).

[99] R. Dijkman, M. Dumas, B. Van Dongen, R. Krik, and J. Mendling, "Similarity of business process models: Metrics and evaluation", *Information Systems*, volume 36, pages 498–516, 2011 (cited on page 79).

[100] P. M. Dixit, H. M. W. Verbeek, and W. M. van der Aalst, "Fast conformance analysis based on activity log abstraction", *IEEE 22nd International Enterprise Distributed Object Computing Conference*, pages 135–144, 2018 (cited on page 79).

[101] J. Dong, D. S. Lad, and Y. Zhao, "Dp-miner: Design pattern discovery using matrix", in *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, IEEE, 2007, pages 371–380 (cited on page 139).

[102] B. F. van Dongen, J. Carmona, T. Chatain, and F. Taymouri, "Aligning modeled and observed behavior: A compromise between computation complexity and quality", in *International Conference on Advanced Information Systems Engineering*, 2017 (cited on page 79).

[103] R. Doornbos, J. Hooman, and B. van Vlimmeren, "Complementary verification of embedded software using asd and uppaal", *International Conference on Innovations in Information Technology*, pages 60–65, 2012 (cited on page 162).

[104] F. Doray and M. Dagenais, "Diagnosing Performance Variations by Comparing Multi-Level Execution Traces", *IEEE Transactions on Parallel and Distributed Systems*, volume 28, pages 462–474, 2017 (cited on pages 79, 81, 83, 86).

[105] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research", in *Guide to advanced empirical software engineering*, Springer, 2008, pages 285–311 (cited on pages 134, 138).

[106] D. Edwards, S. Simmons, and N. Wilde, "An approach to feature location in distributed systems", *Journal of Systems and Software*, volume 79, pages 57–68, 2006 (cited on pages 57, 79, 81, 86, 94, 96, 97).

[107] D. El-Masri, F. Petrillo, Y.-G. Guéhéneuc, A. Hamou-Lhadj, and A. Bouziane, "A systematic literature review on automated log abstraction techniques", *Information and Software Technology*, volume 122, page 106 276, 2020 (cited on pages 60, 105).

[108] A. Etien and C. Salinesi, "Managing requirements in a co-evolution context", in *13th IEEE International Conference on Requirements Engineering*, IEEE, 2005, pages 125–134 (cited on page 59).

[109] *Fdr homepage*, http://www.fsel.com. Accessed: November 2018 (cited on page 137).

[110] D. Fahland and W. M. van der Aalst, "Model repair - Aligning process models to reality", *Information Systems*, volume 47, pages 220–243, 2015 (cited on page 79).

[111] K. Farias, A. Garcia, J. Whittle, and C. Lucena, "Analyzing the effort of composing design models of large-scale software in industrial case studies", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2013, pages 639–655 (cited on page 163).

[112] Y. Feng, K. Dreef, J. A. Jones, and A. van Deursen, "Hierarchical abstraction of execution traces for program comprehension", *IEEE/ACM 26th International Conference on Program Comprehension*, pages 86–8610, 2018 (cited on page 59).

[113] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering", 1987 (cited on pages 57, 97).

[114] R. A. Fisher, "On the interpretation of χ 2 from contingency tables, and the calculation of p", *Journal of the Royal Statistical Society*, volume 85, pages 87–94, 1922 (cited on page 142).

[115] S. D. Fleming and R. Stirewalt, *Successful strategies for debugging concurrent software: an empirical investigation*. Michigan State University. Computer Science, 2009 (cited on page 58).

[116] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling:tree differencing for fine-grained source code change extraction", *IEEE Transactions on Software Engineering*, volume 33, 2007 (cited on page 57).

[117] B. Flyvbjerg, *Five Misunderstandings about Case-Study Research*. Sage, 2007, pages 390–404 (cited on page 15).

[118] F. Folino, G. Greco, A. Guzzo, and L. Pontieri, "Mining usage scenarios in business processes: Outlier-aware discovery and run-time prediction", *Data and Knowledge Engineering*, volume 70, pages 1005–1029, 2011 (cited on page 79).

[119] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework", in *Symposium on Networked Systems Design and Implementation*, 2007 (cited on pages 97, 103).

[120] S. Frenkel and V. Zakharov, *The Conception of Strings Similarity in Software Engineering*. Springer International Publishing, 2021, volume 1288 CCIS, pages 56–67 (cited on page 79).

[121] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis", in *Ninth IEEE international conference on data mining*, IEEE, 2009, pages 149–158 (cited on pages 79, 81, 83, 86, 96, 102).

[122] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie, "Where do developers log? an empirical study on logging practices in industry", *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014 (cited on pages 26, 48, 54).

[123] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models", *IEEE Transactions on software engineering*, volume 17, pages 591–603, 1991 (cited on page 114).

[124] J. Gait, "A probe effect in concurrent programs", *Software: Practice and Experience*, volume 16, pages 225–233, 1986 (cited on page 45).

[125] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design", in *European Conference on Object-Oriented Programming*, Springer, 1993, pages 406–431 (cited on pages 161, 162).

[126] D. Gao, M. K. Reiter, and D. Song, "Beyond output voting: Detecting compromised replicas using hmm-based behavioral distance", *IEEE Transactions on Dependable and Secure Computing*, volume 6, pages 96–110, 2008 (cited on page 79).

[127] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques", in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, 2013, pages 486–496 (cited on page 139).

[128] V. Garousi, M. Felderer, Ç. M. Karapıçak, and U. Yılmaz, "Testing embedded software: A survey of the literature", *Information and Software Technology*, volume 104, pages 14–45, 2018 (cited on page 2).

[129] V. Garousi, K. Petersen, and B. Ozkan, "Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review", *Information and Software Technology*, volume 79, pages 106–127, 2016 (cited on pages 103, 175).

[130] L. Genga, M. Alizadeh, D. Potena, C. Diamantini, and N. Zannone, "Discovering anomalous frequent patterns from partially ordered event logs", *Journal of Intelligent Information Systems*, volume 51, pages 257–300, 2018 (cited on page 79).

[131] C. Gerth, M. Luckey, J. M. Küster, and G. Engels, "Precise mappings between business process models in versioning scenarios", *IEEE International Conference on Services Computing*, pages 218–225, 2011 (cited on page 79).

[132] S. Gholamian and P. A. Ward, "A comprehensive survey of logging in software: From logging statements automation to log mining and analysis", *arXiv preprint arXiv:2110.12489*, 2021 (cited on pages 47, 49).

[133] S. Gholamian and P. A. Ward, "What distributed systems say: A study of seven spark application logs", in *40th International Symposium on Reliable Distributed Systems*, IEEE, 2021, pages 222–232 (cited on pages 48, 54).

[134] E. M. Gold, "Language identification in the limit", *Information and control*, volume 10, pages 447–474, 1967 (cited on pages 6, 111, 116).

[135] M. Goldstein, D. Raz, and I. Segall, "Experience report: Log-based behavioral differencing", in *28th International Symposium on Software Reliability Engineering*, IEEE, 2017, pages 282–293 (cited on pages 56, 79, 81, 83, 86, 90).

[136] C. A. González and J. Cabot, "Formal verification of static software models in MDE: A systematic review", *Information and Software Technology*, volume 56, pages 821–838, 2014 (cited on page 110).

[137] J. C. Gower and G. J. Ross, "Minimum spanning trees and single linkage cluster analysis", *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, volume 18, pages 54–64, 1969 (cited on page 89).

[138] B. Graaf, M. Lormans, and H. Toetenel, "Embedded software engineering: The state of the practice", *IEEE Software*, volume 20, pages 61–69, 2003 (cited on pages 1, 2, 12, 57, 62, 159).

[139] G. Greco, A. Guzzo, L. Pontieri, and D. Saccá, "Discovering expressive process models by clustering log traces", *IEEE Transactions on Knowledge and Data Engineering*, volume 18, pages 1010–1027, 2006 (cited on page 79).

[140] T. Greifenberg, K. Hölldobler, C. Kolassa, M. Look, P. M. S. Nazari, K. Müller, A. N. Perez, D. Plotnikov, D. Reiss, A. Roth, *et al.*, "Integration of handwritten and generated object-oriented code", in *International Conference on Model-Driven Engineering and Software Development*, Springer, 2015, pages 112–132 (cited on page 161).

[141] M. Greiler, A. van Deursen, and A. Zaidman, "Measuring test case similarity to support test suite understanding", in *50th International Conference on Objects, Models, Components, Patterns*, Springer, 2012, pages 91–107 (cited on pages 79, 82, 84, 86, 91).

[142] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics", *International Journal on Software Tools for Technology Transfer*, volume 8, pages 229–247, 2006 (cited on pages 79, 86, 88, 91).

[143] A. Groce and W. Visser, "What went wrong: Explaining counterexamples", *Model Checking Software)*, volume 2648, pages 121–135, 2003 (cited on pages 79, 86).

[144] S. Gu, G. Rong, H. Zhang, and H. Shen, "Logging practices in software engineering: A systematic mapping study", *IEEE Transactions on Software Engineering*, 2022 (cited on page 53).

[145] Z. Gu, E. T. Barr, D. Schleck, and Z. Su, "Reusing debugging knowledge via trace-based bug search", *ACM SIGPLAN Notices*, volume 47, pages 927–942, 2012 (cited on pages 79, 81, 82, 86, 88).

[146] M. A. Gulzar, Y. Zhu, and X. Han, "Perception and practices of differential testing", in *41st International Conference on Software Engineering: Software Engineering in Practice*, IEEE, 2019, pages 71–80 (cited on pages 5, 51–53, 65, 66, 68, 78, 99, 175).

[147] L. Guo, A. Roychoudhury, and T. Wang, "Accurately choosing execution runs for software fault localization", in *15th International Conference on Compiler Construction*, Springer, 2006, pages 80–95 (cited on pages 79, 86).

[148] M. Gupta, A. Mandal, G. Dasgupta, and A. Serebrenik, "Runtime monitoring in continuous deployment by differencing execution behavior model", in *International Conference on Service Oriented Computing*, 2018 (cited on pages 54, 162).

[149] A. Hamou-Lhadj, S. S. Murtaza, W. Fadel, A. Mehrabian, M. Couture, and R. Khoury, "Software behaviour correlation in a redundant and diverse environment using the concept of trace abstraction", in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, 2013, pages 328–335 (cited on page 79).

[150] A. Hannousse, "Searching relevant papers for software engineering secondary studies: Semantic scholar coverage and identification role", *IET SOFTWARE*, volume 15, pages 126–146, 2021 (cited on page 74).

[151] D. Hao, L. Zhang, Y. Pan, H. Mei, and J. Sun, "On similarity-awareness in testing-based fault localization", *Automated Software Engineering*, volume 15, pages 207–249, 2008 (cited on pages 79, 86, 88, 91).

[152] M. Haran, A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil, and S. Fouche, "Techniques for classifying executions of deployed software to support software engineering tasks", *IEEE Transactions on Software Engineering*, volume 33, pages 287–304, 2007 (cited on pages 79, 81, 86).

[153] J. Harty, H. Zhang, L. Wei, L. Pascarella, M. Aniche, and W. Shang, "Logging practices with mobile analytics: An empirical study on firebase", in *8th International Conference on Mobile Software Engineering and Systems*, IEEE, 2021, pages 56–60 (cited on pages 48, 53, 56).

[154] S. He, P. He, Z. Chen, T. Yang, Y. Su, and M. R. Lyu, "A survey on automated log analysis for reliability engineering", *ACM Computing Surveys*, volume 54, pages 1–37, 2021 (cited on pages 42, 47, 53, 60, 105, 106, 175).

[155] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection", in *27th international symposium on software reliability engineering*, IEEE, 2016, pages 207–218 (cited on pages 79, 86, 89).

[156] X. He, P. Avgeriou, P. Liang, and Z. Li, "Technical debt in mde: A case study on gmf/emf-based projects", in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016, pages 162–172 (cited on page 165).

[157] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, "The quest for open source projects that use uml: Mining github", in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, 2016, pages 173–183 (cited on page 165).

[158] D. Hendriks and K. Aslam, "A systematic approach for interfacing component-based software with an active automata learning tool", in *Leveraging Applications of Formal Methods*, 2022 (cited on page 172).

[159] D. Hendriks, A. v. d. Meer, and W. Oortwijn, "A multi-level methodology for behavioral comparison of software-intensive systems", in *International Conference on Formal Methods for Industrial Critical Systems*, Springer, 2022, pages 226–243 (cited on pages 61, 101, 169, 174).

[160] M. Hendriks, J. Verriet, T. Basten, B. Theelen, M. Brassé, and L. Somers, "Analyzing execution traces: Critical-path analysis and distance analysis", *International Journal on Software Tools for Technology Transfer*, volume 19, pages 487–510, 2017 (cited on page 79).

[161] T. A. Henzinger and J. Sifakis, "The discipline of embedded systems design", *Computer*, volume 40, pages 32–40, 2007 (cited on page 57).

[162] J. D. Herbsleb and E. Kuwana, "Preserving knowledge in design projects: What designers need to know", *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, 1993 (cited on page 105).

[163] K. Hoffman, P. Eugster, and S. Jagannathan, "Semantics-aware trace analysis", *ACM SIGPLAN Notices*, volume 44, pages 453–464, 2009 (cited on pages 79, 83, 86, 87, 94–96).

[164] J. A. Holton, "The coding process and its challenges", *The Sage handbook of grounded theory*, volume 3, pages 265–289, 2007 (cited on page 17).

[165] B. Hompes, J. C. Buijs, W. M. van der Aalst, P. M. Dixit, and J. Buurman, "Detecting changes in process behavior using comparative case clustering", in *5th IFIP WG 2.6 International Symposium on Data-Driven Process Discovery and Analysis*, Springer, 2017, pages 54–75 (cited on page 79).

[166] B. Hooimeijer, M. Geilen, J. F. Groote, D. Hendriks, and R. Schiffelers, "Constructive Model Inference: Model learning for component-based software architectures", in *17th International Conference on Software Technologies*, 2022, pages 146–158 (cited on pages 61, 101).

[167] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman, 2006 (cited on page 121).

[168] F. Howar, D. Giannakopoulou, and Z. Rakamarić, "Hybrid learning: Interface generation through static, dynamic, and symbolic analysis", in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ACM, 2013, pages 268–279 (cited on pages 130, 132).

[169] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure", *Science of Computer Programming*, volume 89, pages 144–161, 2014 (cited on pages 160, 163, 164, 173, 175).

[170] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of mde in industry", in *Proceedings of the 33rd international conference on software engineering*, ACM, 2011, pages 471–480 (cited on pages 161, 163, 164).

[171] *Ieee xplore digital library*, `https://ieeexplore.ieee.org/Xplore/home.jsp`. Accessed: April 2021 (cited on page 69).

[172] M. Idris, A. Mehrabian, A. Hamou-Lhadj, and R. Khoury, "Pattern-based trace correlation technique to compare software versions", in *Third International Conference on Autonomous and Intelligent Systems*, Springer, 2012, pages 159–166 (cited on pages 79, 81, 84, 86, 88).

[173] H. Ikeuchi, A. Watanabe, T. Kawata, and R. Kawahara, "Root-cause diagnosis using logs generated by user actions", in *IEEE Global Communications Conference*, IEEE, 2018, pages 1–7 (cited on pages 79, 86).

[174] M. Isberner, F. Howar, and B. Steffen, "The TTT algorithm: A redundancy-free approach to active automata learning", in *International Conference on Runtime Verification*, Springer, 2014, pages 307–322 (cited on pages 110, 112, 117).

[175] M. Isberner, "Foundations of active automata learning: An algorithmic perspective", Ph.D. dissertation, 2015 (cited on page 131).

[176] M. Ishrat, M. Saxena, and M. Alamgir, "Comparison of static and dynamic analysis for runtime monitoring", *International Journal of Computer Science & Communication Networks*, volume 2, 2012 (cited on page 6).

[177] *Jaeger 2022*, `https://www.jaegertracing.io/`. Accessed: September 2022 (cited on page 103).

[178] M. Jasper, M. Mues, A. Murtovi, M. Schlüter, F. Howar, B. Steffen, M. Schordan, D. Hendriks, R. Schiffelers, H. Kuppens, *et al.*, *RERS 2019: combining synthesis with real-world models*. Springer, 2019 (cited on page 132).

[179] D. F. Jerding, J. T. Stasko, and T. Ball, "Visualizing interactions in program executions", *Proceedings of the 19th International Conference on Software Engineering*, pages 360–370, 1997 (cited on page 59).

[180] W. Jin and A. Orso, "F3: Fault localization for field failures", in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pages 213–223 (cited on page 98).

[181] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential slicing: Identifying causal execution differences for security applications", in *IEEE Symposium on Security and Privacy*, IEEE, 2011, pages 347–362 (cited on page 79).

[182] R. Jolak, T. Ho-Quang, M. R. V. Chaudron, and R. R. H. Schiffelers, "Model-based software engineering: A multiple-case study on challenges and development efforts", in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, 2018, pages 213–223 (cited on page 159).

[183] S. B. Junior, P. Ceravolo, E. Damiani, N. J. Omori, and G. M. Tavares, "Anomaly detection on event logs with a scarcity of labels", in *2nd International Conference on Process Mining*, IEEE, 2020, pages 161–168 (cited on page 86).

[184] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the apache software foundation projects", in *13th Working Conference on Mining Software Repositories*, IEEE, 2016, pages 154–164 (cited on pages 48, 53, 59).

[185] S. Kabinna, C.-P. Bezemer, W. Shang, M. D. Syer, and A. E. Hassan, "Examining the stability of logging statements", *Empirical Software Engineering*, volume 23, pages 290–333, 2018 (cited on pages 48, 59).

[186] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code", in *17th Working Conference on Reverse Engineering*, IEEE, 2010, pages 119–128 (cited on page 56).

[187] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, *et al.*, "Canopy: An end-to-end performance tracing and analysis system", in *Proceedings of the 26th symposium on operating systems principles*, 2017, pages 34–50 (cited on page 103).

[188] H. J. Kang and D. Lo, "Adversarial specification mining", *ACM Transactions on Software Engineering and Methodology*, volume 30, pages 1–40, 2021 (cited on page 132).

[189] U. Kargén and N. Shahmehri, "Towards robust instruction-level trace alignment of binary code", *32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 342–352, 2017 (cited on page 81).

[190] Y. G. Kim, H. S. Hong, D.-H. Bae, and S. D. Cha, "Test cases generation from uml state diagrams", *IEE Proceedings-Software*, volume 146, pages 187–192, 1999 (cited on page 136).

[191] D. Kirat and G. Vigna, "Malgene: Automatic extraction of malware analysis evasion signature", in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pages 769–780 (cited on page 79).

[192] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering", 2007 (cited on page 69).

[193] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams", in *29th International Conference on Software Engineering*, IEEE, 2007, pages 344–353 (cited on page 105).

[194] G. Kremer, P. Owezarski, and P. Berthou, "Cross fertilization between wireless testbeds and ns-3 simulation models", in *Proceedings of the 20th ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems*, 2017, pages 295–302 (cited on pages 79, 80).

[195] I. Krka, Y. Brun, and N. Medvidovic, "Automatic mining of specifications from invocation traces and method invariants", in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pages 178–189 (cited on page 58).

[196] A. Kuhn, S. Ducasse, and T. Gírba, "Semantic clustering: Identifying topics in source code", *Information and Software Technology*, volume 49, pages 230–243, 2007 (cited on page 139).

[197] T. R. Kurfess and T. J. Hodgson, "Metrology, sensors and control", in *Micromanufacturing*, Springer, 2007, pages 89–109 (cited on page 15).

[198] A. P. Kurniati, C. McInerney, K. Zucker, G. Hall, D. Hogg, and O. Johnson, "Using a multi-level process comparison for process change analysis in cancer pathways", *International Journal of Environmental Research and Public Health*, volume 17, pages 1–16, 2020 (cited on page 79).

[199] M. La Rosa, M. Dumas, C. C. Ekanayake, L. García-Bañuelos, J. Recker, and A. H. ter Hofstede, "Detecting approximate clones in business process model repositories", *Information Systems*, volume 49, pages 102–125, 2015 (cited on page 165).

[200] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting", in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pages 101–111 (cited on pages 79, 86, 92, 94, 96, 97).

[201] B. Lambeau, C. Damas, and P. Dupont, "State-merging DFA induction algorithms with mandatory merge constraints", in *International Colloquium on Grammatical Inference*, Springer, 2008, pages 139–153 (cited on pages 115, 116).

[202] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study", in *IEEE/ACM 39th International Conference on Software Engineering*, IEEE, 2017, pages 507–518 (cited on pages 70, 71).

[203] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm", in *International Colloquium on Grammatical Inference*, Springer, 1998, pages 1–12 (cited on page 111).

[204] C. F. Lange and M. R. V. Chaudron, "An empirical assessment of completeness in uml designs", in *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering*, IET, 2004, pages 111–121 (cited on page 164).

[205] C. F. Lange and M. R. V. Chaudron, "Effects of defects in uml models: An experimental investigation", in *Proceedings of the 28th international conference on Software engineering*, ACM, 2006, pages 401–411 (cited on page 7).

[206] C. F. Lange, M. R. V. Chaudron, and J. Muskens, "In practice: Uml software architecture and design description", *IEEE software*, volume 23, pages 40–46, 2006 (cited on page 160).

[207] C. F. Lange, B. DuBois, M. R. V. Chaudron, and S. Demeyer, "An experimental investigation of uml modeling conventions", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2006, pages 27–41 (cited on page 165).

[208] J. R. Larus, "Whole program paths", *ACM SIGPLAN Notices*, volume 34, pages 259–269, 1999 (cited on page 97).

[209] T.-D. B. Le, D. Lo, C. L. Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants", *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016 (cited on pages 79, 81, 86, 91).

[210] T.-D. B. Le and D. Lo, "Deep specification mining", in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pages 106–117 (cited on page 132).

[211] D. Lee, S. K. Cha, and A. H. Lee, "A performance anomaly detection and analysis framework for DBMS development", *IEEE Transactions on Knowledge and Data Engineering*, volume 24, pages 1345–1360, 2012 (cited on page 79).

[212] E. A. Lee, "Cyber physical systems: Design challenges", in *11th IEEE international symposium on object and component-oriented real-time distributed computing*, IEEE, 2008, pages 363–369 (cited on page 1).

[213] S. J. J. Leemans, K. Goel, and S. J. van Zelst, "Using multi-level information in hierarchical process mining: Balancing behavioural quality and model complexity", *2nd International Conference on Process Mining*, pages 137–144, 2020 (cited on page 79).

[214] S. J. Leemans, W. M. van der Aalst, T. Brockhoff, and A. Polyvyanyy, "Stochastic process mining: Earth movers' stochastic conformance", *Information Systems*, volume 102, page 101 724, 2021 (cited on page 79).

[215] S. J. Leemans, D. Fahland, and W. M. van der Aalst, "Scalable process discovery and conformance checking", *Software and Systems Modeling*, volume 17, pages 599–631, 2018 (cited on pages 79, 88).

[216] S. J. Leemans, S. Shabaninejad, K. Goel, H. Khosravi, S. Sadiq, and M. T. Wynn, "Identifying cohorts: Recommending drill-downs based on differences in behaviour for process mining", in *39th International Conference on Conceptual Modeling*, Springer, 2020, pages 92–102 (cited on page 79).

[217] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java api specifications", *31st IEEE/ACM International Conference on Automated Software Engineering*, pages 602–613, 2016 (cited on page 11).

[218] M. M. Lehman, "Laws of software evolution revisited", in *European Workshop on Software Process Technology*, Springer, 1996, pages 108–124 (cited on page 2).

[219] C. Lemieux, D. Park, and I. Beschastnikh, "General ltl specification mining", in *30th IEEE/ACM International Conference on Automated Software Engineering*, IEEE, 2015, pages 81–92 (cited on pages 58, 162).

[220] G. Leonardi, M. Striani, S. Quaglini, A. Cavallini, and S. Montani, "Leveraging semantic labels for multi-level abstraction in medical process mining and trace comparison", *Journal of Biomedical Informatics*, volume 83, pages 10–24, 2018 (cited on page 79).

[221] O. Levy and D. G. Feitelson, "Understanding large-scale software–a hierarchical view", in *IEEE/ACM 27th International Conference on Program Comprehension*, IEEE, 2019, pages 283–293 (cited on pages 58, 104).

[222] C. Li, M. Reichert, and A. Wombacher, "Discovering reference process models by mining process variants", *IEEE International Conference on Web Services*, pages 45–53, 2008 (cited on page 79).

[223] H. Li, T.-H. P. Chen, W. Shang, and A. Hassan, "Studying software logging using topic models", *Empirical Software Engineering*, volume 23, pages 2655–2694, 2018 (cited on pages 26, 54).

[224] H. Li, W. Shang, B. Adams, M. Sayagh, and A. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives", *IEEE Transactions on Software Engineering*, volume 47, pages 2858–2873, 2020 (cited on pages 12, 14, 19, 48–52, 54, 65).

[225] S. Li, X. Niu, Z. Jia, X. Liao, J. Wang, and T. Li, "Guiding log revisions by learning from software evolution history", *Empirical Software Engineering*, volume 25, pages 2302–2340, 2020 (cited on page 59).

[226] T. Li, J. Ma, Q. Pei, Y. Shen, C. Lin, S. Ma, and M. S. Obaidat, "Aclog: Attack chain construction based on log correlation", in *IEEE Global Communications Conference*, IEEE, 2019, pages 1–6 (cited on pages 79, 80).

[227] X. Lian, M. Rahimi, J. Cleland-Huang, L. Zhang, R. Ferrai, and M. Smith, "Mining requirements knowledge from collections of domain documents", in *IEEE 24th International Requirements Engineering Conference*, 2016, pages 156–165 (cited on page 59).

[228] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Assessing the state-of-practice of model-based engineering in the embedded systems domain", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2014, pages 166–182 (cited on pages 3, 5, 134, 135, 161, 163).

[229] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Model-based engineering in the embedded systems domain: An industrial survey on the state-of-practice", *Software & Systems Modeling*, volume 17, pages 91–113, 2018 (cited on pages 3, 158, 163).

[230] B. Lin, N. Cassee, A. Serebrenik, G. Bavota, N. Novielli, and M. Lanza, "Opinion mining for software development: A systematic literature review", *ACM Transactions on Software Engineering and Methodology*, volume 31, pages 1 –41, 2022 (cited on page 72).

[231] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen, "Log clustering based problem identification for online service systems", *IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111, 2016 (cited on pages 79, 86, 89, 92).

[232] C. Liu and J. Han, "Failure proximity: A fault localization-based approach", in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pages 46–56 (cited on page 79).

[233] Q. Liu, Q. Bai, and Y. Yang, "Semantic similarity of workflow traces with various granularities", in *17th International Conference on Web Information Systems Engineering*, Springer, 2016, pages 211–226 (cited on page 79).

[234] Q. Liu, X. Zhao, K. Taylor, X. Lin, G. Squire, C. Kloppers, and R. Miller, "Towards semantic comparison of multi-granularity process traces", *Knowledge-Based Systems*, volume 52, pages 91–106, 2013 (cited on page 79).

[235] X. Liu, W. Lin, A. Pan, and Z. Zhang, "Wids checker: Combating bugs in distributed systems", in *Symposium on Networked Systems Design and Implementation*, 2007 (cited on pages 57, 81).

[236] X. Liu, M. Alshangiti, C. Ding, and Q. Yu, "Log sequence clustering for workflow mining in multi-workflow systems", *Data and Knowledge Engineering*, volume 117, pages 1–17, 2018 (cited on page 79).

[237] X. Liu, H. Liu, and C. Ding, "Incorporating user behavior patterns to discover workflow models from event logs", *IEEE 20th International Conference on Web Services*, pages 171–178, 2013 (cited on page 79).

[238] D. Lo and S.-C. Khoo, "Quark: Empirical assessment of automaton-based specification miners", *13th Working Conference on Reverse Engineering*, pages 51–60, 2006 (cited on page 111).

[239] D. Lo, L.Mariani, and M. Santoro, "Learning extended FSA from software: An empirical assessment", *Journal of Systems and Software*, volume 85, pages 2063–2076, 2012 (cited on pages 115, 116).

[240] D. Lo and S. Maoz, "Scenario-based and value-based specification mining: Better together", *Automated Software Engineering*, volume 19, pages 423–458, 2010 (cited on page 58).

[241] D. Lo, S.-C. Khoo, J. Han, and C. Liu, *Mining software specifications: methodologies and applications*. CRC Press, 2011 (cited on page 162).

[242] D. Lo, S.-C. Khoo, and C. Liu, "Efficient mining of iterative patterns for software specification discovery", in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pages 460–469 (cited on pages 79, 86).

[243] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference", in *Proceedings of the 7th Joint Meeting Of The European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pages 345–354 (cited on page 58).

[244] W. Z. Low, W. M. van der Aalst, A. H. ter Hofstede, M. T. Wynn, and J. De Weerdt, "Change visualisation: Analysing the resource and timing differences between two event logs", *Information Systems*, volume 65, pages 106–123, 2017 (cited on pages 79, 80).

[245] Y. Lu, T. Nolte, I. Bate, J. Kraft, and C. Norström, "Assessment of trace-differences in timing analysis for complex real-time embedded systems", *6th IEEE International Symposium on Industrial and Embedded Systems*, pages 284–293, 2011 (cited on pages 79, 81, 86, 89).

[246] Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software", *IEEE/ACM 13th Working Conference on Mining Software Repositories*, pages 25–36, 2016 (cited on page 91).

[247] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests", *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014 (cited on pages 21, 24).

[248] A. Maaradji, M. Dumas, M. L. Rosa, and A. Ostovar, "Fast and accurate business process drift detection", in *International Conference on Business Process Management*, 2015 (cited on page 79).

[249] A. Maaradji, M. Dumas, M. L. Rosa, and A. Ostovar, "Detecting sudden and gradual drifts in business processes from execution traces", *IEEE Transactions on Knowledge and Data Engineering*, volume 29, pages 2140–2154, 2017. arXiv: 2005.04016 (cited on page 79).

[250] A. MacDonald, D. Russell, and B. Atchison, "Model-driven development within a legacy system: An industry experience report", in *Australian Software Engineering Conference*, IEEE, 2005, pages 14–22 (cited on page 159).

[251] J. Maeyens, A. Vorstermans, and M. Verbeke, "Process mining on machine event logs for profiling abnormal behaviour and root cause analysis", *Annals of Telecommunications*, volume 75, pages 563 –572, 2020 (cited on page 79).

[252] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis", *IEEE Transactions on Dependable and Secure Computing*, volume 7, pages 381–395, 2010 (cited on page 79).

[253] S. Maoz and D. Harel, "On tracing reactive systems", *Software and Systems Modeling*, volume 10, pages 447–468, 2011 (cited on pages 79, 81, 86, 91).

[254] S. Maoz, J. O. Ringert, and B. Rumpe, "A manifesto for semantic model differencing", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2010, pages 194–203 (cited on pages 4, 11, 56).

[255] T. Margaria, O. Niese, H. Raffelt, and B. Steffen, "Efficient test-based model generation for legacy reactive systems", in *High–Level Design Validation and Test Workshop*, IEEE, 2004, pages 95–100 (cited on page 114).

[256] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs", in *19th International Symposium on Software Reliability Engineering*, IEEE, 2008, pages 117–126 (cited on pages 79, 81, 83, 86, 90).

[257] J. A. Martínez-Carrascal, E. Valderrama, and T Sancho-Vinuesa, "Combining clustering and sequential pattern mining to detect behavioral differences in log data: Conceptualization and case study", 2020 (cited on pages 79, 80).

[258] M. J. Mashhadi and H. Hemmati, "An empirical study on practicality of specification mining algorithms on a real-world application", *IEEE/ACM 27th International Conference on Program Comprehension*, pages 65–69, 2019 (cited on pages 11, 12).

[259] M. J. Mashhadi, T. R. Siddiqui, H. Hemmati, and H. Loewen, "Interactive semi-automated specification mining for debugging: An experience report", *Information and Software Technology*, volume 113, pages 20–38, 2019 (cited on pages 58, 101).

[260] F. Mattern *et al.*, *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988 (cited on pages 57, 97).

[261] P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, "Improving topic model source code summarization", in *Proceedings of the 22nd international conference on program comprehension*, 2014, pages 291–294 (cited on page 59).

[262] M. L. McHugh, "Interrater reliability: The kappa statistic", *Biochemia medica*, volume 22, pages 276–282, 2012 (cited on page 73).

[263] L. Meng, F. Ji, Y. Sun, and T. Wang, "Detecting anomalies in microservices with execution trace comparison", *Future Generation Computer Systems*, volume 116, pages 291–301, 2021 (cited on pages 79, 81, 86, 87).

[264] J. G. M. Mengerink, A. Serebrenik, R. R. H. Schiffelers, and M. G. J. van den Brand, "Automated analyses of model-driven artifacts: Obtaining insights into industrial application of mde", in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, ACM, 2017, pages 116–121 (cited on pages 141, 142).

[265] J. Mengerink, R. R. H. Schiffelers, A. Serebrenik, and M. van den Brand, "Dsl/model co-evolution in industrial emf-based MDSE ecosystems", in *Proceedings of the 10th Workshop on Models and Evolution*, volume 1706, 2016, pages 2–7 (cited on page 59).

[266] G. K. Michelon, L. Linsbauer, W. K. Assunção, S. Fischer, and A. Egyed, "A hybrid feature location technique for re-engineeringsingle systems into software product lines", in *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, 2021, pages 1–9 (cited on pages 79, 81, 86, 87).

[267] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking", in *Proceedings of the 26th USENIX Security Symposium*, 2017 (cited on page 79).

[268] A. V. Miranskyy, M. Davison, R. M. Reesor, and S. S. Murtaza, "Using entropy measures for comparison of software traces", *Information Sciences*, volume 203, pages 59–72, 2012. arXiv: 1010.5537 (cited on pages 79, 81, 86).

[269] A. V. Miranskyy, N. H. Madhavji, M. S. Gittens, M. Davison, M. Wilding, D. Godwin, and C. A. Taylor, "Sift: A scalable iterative-unfolding technique for filtering execution traces", in *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, 2008, pages 274–288 (cited on pages 79, 81, 82, 86, 94, 95).

[270] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments", in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, 2006, 88–es (cited on pages 79, 81, 86, 88, 91).

[271] A. V. Mirgorodskiy and B. P. Miller, "Diagnosing distributed systems with self-propelled instrumentation", in *International Middleware Conference*, 2008 (cited on pages 79, 86, 90, 94, 96, 97, 102).

[272] D. Mishra, A. Mishra, and A. Yazici, "Successful requirement elicitation by combining requirement engineering techniques", in *First International Conference on the Applications of Digital Information and Web Technologies*, IEEE, 2008, pages 258–263 (cited on page 55).

[273] P. Mohagheghi and V. Dehlen, "Where is the proof? - a review of experiences from applying mde in industry", in *European Conference on Model Driven Architecture-Foundations and Applications*, Springer, 2008, pages 432–443 (cited on pages 159, 163).

[274] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani, "Mde adoption in industry: Challenges and success criteria", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2008, pages 54–59 (cited on page 5).

[275] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases", *Empirical Software Engineering*, volume 18, pages 89–116, 2013 (cited on page 163).

[276] K. Mohror and K. L. Karavanic, "Trace profiling: Scalable event tracing on high-end parallel systems", *Parallel Computing*, volume 38, pages 194–225, 2012 (cited on pages 79, 81, 83, 86, 88).

[277] S. Montani, G. Leonardi, S. Quaglini, A. Cavallini, and G. Micieli, "A knowledge-intensive approach to process similarity calculation", *Expert Systems with Applications*, volume 42, pages 4207–4215, 2015 (cited on page 79).

[278] S. Montani, G. Leonardi, M. Striani, S. Quaglini, and A. Cavallini, "Multi-level abstraction for trace comparison and process discovery", *Expert Systems with Applications*, volume 81, pages 398–409, 2017 (cited on page 79).

[279] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection", *ACM Transactions on Information and System Security*, volume 9, pages 61–93, 2006 (cited on page 79).

[280] T. Noergaard, *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012 (cited on page 22).

[281] T. Nolle, S. Luettgen, A. Seeliger, and M. Mühlhäuser, "Analyzing business process anomalies using autoencoders", *Machine Learning*, volume 107, pages 1875–1893, 2018. arXiv: 1803 . 01092 (cited on page 79).

[282] A. Nurwidyantoro, T. Ho-Quang, and M. R. V. Chaudron, "Automated classification of class role-stereotypes via machine learning", in *Proceedings of the Evaluation and Assessment on Software Engineering*, ACM, 2019, pages 79–88 (cited on page 139).

[283] F. S. Ocariza and B. Zhao, "Localizing software performance regressions in web applications by comparing execution timelines", *Software Testing Verification and Reliability*, volume 31, pages 1–32, 2021 (cited on pages 79, 81, 86, 87).

[284] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun, "Behavioral resource-aware model inference", in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pages 19–30 (cited on pages 79, 83, 86, 92, 93).

[285] J. Oncina and P. García, "Identifying regular languages in polynomial time", 1993 (cited on pages 115, 123).

[286] J. Osborn, B. Samuel, J. McCoy, and M. Mateas, "Evaluating play trace (dis) similarity metrics", in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 10, 2014, pages 139–145 (cited on page 79).

[287] M. H. Osman, T. Ho-Quang, and M. R. V. Chaudron, "An automated approach for classifying reverse-engineered and forward-engineered uml class diagrams", in *44th Euromicro Conference on Software Engineering and Advanced Applications*, IEEE, 2018, pages 396–399 (cited on page 165).

[288] A. Ostovar, S. J. Leemans, and M. L. Rosa, "Robust drift characterization from event streams of business processes", *ACM Transactions on Knowledge Discovery from Data*, volume 14, 2020 (cited on page 79).

[289] A. Ostovar, A. Maaradji, M. La Rosa, and A. H. ter Hofstede, "Characterizing drift from event streams of business processes", in *29th International Conference on Advanced Information Systems Engineering*, Springer, 2017, pages 210–228 (cited on page 79).

[290] K. Otomo, S. Kobayashi, K. Fukuda, and H. Esaki, "Latent semantics approach for network log analysis: Modeling and its application", in *International Symposium on Integrated Network Management*, IEEE, 2021, pages 215–223 (cited on page 79).

[291] F. Palomba, D. A. Tamburri, F. Arcelli Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik, "Beyond technical aspects: How do community smells influence the intensity of code smells?", *IEEE Transactions on Software Engineering*, pages 1–1, 2018 (cited on pages 7, 134).

[292] D. Pandey, U. Suman, and A. K. Ramani, "An effective requirement engineering process model for software development and requirements management", in *International Conference on Advances in Recent Technologies in Communication and Computing*, IEEE, 2010, pages 287–291 (cited on pages 53, 55).

[293] L. Pascarella, D. Spadini, F. Palomba, M. Bruntink, and A. Bacchelli, "Information needs in contemporary code review", *Proceedings of the ACM on Human-Computer Interaction*, volume 2, pages 1–27, 2018 (cited on page 105).

[294] F. Pastore, L. Mariani, A. Goffi, M. Oriol, and M. Wahler, "Dynamic analysis of upgrades in c/c++ software", in *23rd International Symposium on Software Reliability Engineering*, IEEE, 2012, pages 91–100 (cited on pages 79, 86, 90, 94, 99).

[295] A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, "Industry practices and event logging: Assessment of a critical software development process", *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 169–178, 2015 (cited on pages 48, 49, 51–55).

[296] D. Peled, M. Y. Vardi, and M. Yannakakis, "Black box checking", *Journal of Automata, Languages and Combinatorics*, volume 7, pages 225–246, 2002 (cited on page 114).

[297] W. Penson, E. Huang, D. Klamut, E. Wardle, G. Douglas, S. Fazackerley, and R. Lawrence, "Continuous integration platform for arduino embedded software", in *30th Canadian Conference on Electrical and Computer Engineering*, IEEE, 2017, pages 1–4 (cited on page 2).

[298] A. Petrenko, S. Boroday, and R. Groz, "Confirming configurations in efsm testing", *IEEE Transactions on Software engineering*, volume 30, pages 29–42, 2004 (cited on page 136).

[299] J. Pflug and S. Rinderle-Ma, "Process instance similarity: Potentials, metrics, applications", in *On the Move to Meaningful Internet Systems: OTM 2016 Conferences: Confederated International Conferences: CoopIS, CTC, and ODBASE*, Springer, 2016, pages 136–154 (cited on page 79).

[300] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto, "How does code readability change during software evolution?", *Empirical Software Engineering*, pages 1–39, 2020 (cited on page 164).

[301] A. Pini, R. A. Brown, and M. T. Wynn, "Process visualization techniques for multi-perspective process comparisons", in *Asia Pacific Business Process Management*, 2015 (cited on page 79).

[302] F. A. P. Pinto, U. Kulesza, and C. Treude, "Automating the performance deviation analysis for multiple system releases: An evolutionary study", *IEEE 15th International Working Conference on Source Code Analysis and Manipulation*, pages 201–210, 2015 (cited on pages 79, 81, 86, 92, 94, 99).

[303] A. Polyvyanyy and A. Kalenkova, "Conformance checking of partially matching processes: An entropy-based approach", *Information Systems*, volume 106, 2022 (cited on page 79).

[304] A. Polyvyanyy, A. Solti, M. Weidlich, C. D. Ciccio, and J. Mendling, "Monotone precision and recall measures for comparing executions and specifications of dynamic systems", *ACM Transactions on Software Engineering and Methodology*, volume 29, pages 1–41, 2020 (cited on page 79).

[305] E. Poppe, M. T. Wynn, A. H. M. ter Hofstede, R. A. Brown, A. Pini, and W. M. van der Aalst, "Processprofiler3d: A tool for visualising performance differences between process cohorts and process instances", in *International Conference on Business Process Management*, 2017 (cited on page 79).

[306] P. Pourali and J. M. Atlee, "An empirical investigation to understand the difficulties and challenges of software modellers when using modelling tools", in *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, ACM, 2018, pages 224–234 (cited on pages 7, 163, 164).

[307] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines", in *IEEE International Conference on Software Maintenance*, IEEE, 2010, pages 1–10 (cited on pages 79, 81, 86, 88, 132).

[308] M. Pradel and T. R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives", *34th International Conference on Software Engineering*, pages 288–298, 2012 (cited on pages 4, 11).

[309] S. Prochnow, "Efficient development of complex statecharts", Ph.D. dissertation, Christian-Albrechts Universität Kiel, 2008 (cited on page 134).

[310] D. Qi, A. Roychoudhury, and Z. Liang, "DARWIN: An approach to debugging evolving programs", *ACM Transactions on Software Engineering and Methodology*, volume 21, 2012 (cited on pages 79, 86, 94, 98).

[311] H. Raffelt, B. Steffen, T. Berg, and T. Margaria, "LearnLib: A framework for extrapolating behavioral models", *International journal on software tools for technology transfer*, volume 11, page 393, 2009 (cited on page 121).

[312] A. Raghuraman, T. Ho-Quang, M. R. V. Chaudron, A. Serebrenik, and B. Vasilescu, "Does uml modeling associate with lower defect proneness?: A preliminary empirical investigation", in *IEEE/ACM 16th International Conference on Mining Software Repositories*, IEEE, 2019, pages 101–104 (cited on pages 162, 165).

[313] K. Rahad, O. Badreddin, and S. Mohsin Reza, "The human in model-driven engineering loop: A case study on integrating handwritten code in model-driven engineering repositories", *Software: Practice and Experience*, 2021 (cited on pages 165, 176).

[314] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Sieve: A tool for automatically detecting variations across program versions", in *21st International Conference on Automated Software Engineering*, IEEE, 2006, pages 241–252 (cited on pages 79, 84, 86, 87, 91).

[315] V. P. Ranganath, P. Vallathol, and P. Gupta, "Compatibility testing using patterns-based trace comparison", *29th ACM/IEEE International Conference on Automated Software Engineering*, pages 469–478, 2014 (cited on pages 79, 86, 90, 92).

[316] Á. Rebuge and D. R. Ferreira, "Business process analysis in healthcare environments: A methodology based on process mining", *Information Systems*, volume 37, pages 99–116, 2012 (cited on page 79).

[317] D. Reißner, R. Conforti, M. Dumas, M. La Rosa, and A. Armas-Cervantes, "Scalable conformance checking of business processes", in *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE*, Springer, 2017, pages 607–627 (cited on page 79).

[318] C. K. Riemenschneider, B. C. Hardgrave, and F. D. Davis, "Explaining software developer acceptance of methodologies: A comparison of five theoretical models", *IEEE transactions on Software Engineering*, volume 28, pages 1135–1145, 2002 (cited on page 175).

[319] G. Robles, T. Ho-Quang, R. Hebig, M. R. V. Chaudron, and M. A. Fernandez, "An extensive dataset of uml models in github", in *14th International Conference on Mining Software Repositories*, IEEE, 2017, pages 519–522 (cited on page 165).

[320] R. Ronald and S. Robert, "Inference of finite automata using homing sequences", *Information and Computation*, volume 103, pages 299–347, 1993 (cited on page 110).

[321] G. Rong, S. Gu, H. Zhang, D. Shao, and W. Liu, "How is logging practice implemented in open source software projects? a preliminary exploration", in *25th Australasian Software Engineering Conference*, IEEE, 2018, pages 171–180 (cited on pages 48, 51).

[322] G. Rong, Y. Xu, S. Gu, H. Zhang, and D. Shao, "Can you capture information as you intend to? a case study on logging practice in industry", in *International Conference on Software Maintenance and Evolution*, IEEE, 2020, pages 12–22 (cited on pages 48, 50, 52, 53).

[323] G. Rong, Q. Zhang, X. Liu, and S. Gu, "A systematic review of logging practice in software engineering", in *24th Asia-Pacific Software Engineering Conference*, IEEE, 2017, pages 534–539 (cited on page 55).

[324] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering", *Empirical software engineering*, volume 14, page 131, 2009 (cited on pages 2, 14, 121, 138, 162).

[325] W. Said, J. Quante, and R. Koschke, "Towards interactive mining of understandable state machine models from embedded software.", in *MODELSWARD*, 2018, pages 117–128 (cited on pages 4, 11).

[326] A. Sailer, M. Deubzer, G. Lüttgen, and J. Mottok, "Comparing trace recordings of automotive real-time software", in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, 2017, pages 118–127 (cited on pages 79, 86).

[327] H. Saissi, S. Winter, O. Schwahn, K. Pattabiraman, and N. Suri, "Tracesanitizer-eliminating the effects of non-determinism on error propagation analysis", in *50th International Conference on Dependable Systems and Networks*, IEEE, 2020, pages 52–63 (cited on pages 79, 81, 86, 94, 96, 97, 102).

[328] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows", in *Symposium on Networked Systems Design and Implementation*, 2011 (cited on pages 79, 81, 86, 91, 94, 96, 98).

[329] J. de San Pedro and J. Cortadella, "Mining structured petri nets for the visualization of process behavior", in *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*, S. Ossowski, Ed., ACM, 2016, pages 839–846 (cited on page 134).

[330] L. Sanchez, "Learning software behavior through active automata learning with data", M.S. thesis, Eindhoven University of Technology, 2018 (cited on pages 132, 176).

[331] G. Schaefer, "Statechart style checking–automated semantic robustness analysis of statecharts", Ph.D. dissertation, Diploma thesis, Christian-Albrechts-Universität zu Kiel, Institut für Informatik, 2006 (cited on page 134).

[332] R. R. Schiffelers, Y. Luo, J. Mengerink, and M. van den Brand, "Towards automated analysis of model-driven artifacts in industry.", in *MODELSWARD*, 2018, pages 743–751 (cited on pages 2, 6).

[333] *Sciencedirect*, `https://www.sciencedirect.com/`. Accessed: April 2021 (cited on page 69).

[334] *Scopus*, `https://www.scopus.com/home.uri`. Accessed: April 2021 (cited on page 69).

[335] C. B. Seaman, "Qualitative methods in empirical studies of software engineering", *IEEE Transactions on software engineering*, volume 25, pages 557–572, 1999 (cited on page 32).

[336] *Semantic scholar*, `https://www.semanticscholar.org/`. Accessed: July 2021 (cited on page 74).

[337] A. Senderovich, M. Weidlich, L. Yedidsion, A. Gal, A. Mandelbaum, S. Kadish, and C. A. Bunnell, "Conformance checking and performance improvement in scheduled processes: A queueing-network perspective", *Information Systems*, volume 62, pages 185–206, 2016 (cited on page 79).

[338] E. Seo, M. M. H. Khan, P. Mohapatra, J. Han, and T. F. Abdelzaher, "Exposing complex bug-triggering conditions in distributed systems via graph mining", *International Conference on Parallel Processing*, pages 186–195, 2011 (cited on pages 79, 86, 94, 96, 97).

[339] T. M. Shaft and I. Vessey, "Research report - the relevance of application domain knowledge: The case of computer program comprehension", *Information systems research*, volume 6, pages 286–299, 1995 (cited on page 58).

[340] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge", in *International conference on software maintenance and evolution*, IEEE, 2014, pages 21–30 (cited on pages 48, 53, 56, 57).

[341] S. Sharma, A. Hussain, and H. Saran, "Towards repeatability & verifiability in networking experiments: A stochastic framework", *Journal of Network and Computer Applications*, volume 81, pages 12–23, 2017 (cited on page 79).

[342] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, "The role of replications in empirical software engineering", *Empirical software engineering*, volume 13, pages 211–218, 2008 (cited on pages 29, 30).

[343] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure", 2010 (cited on pages 97, 103).

[344] A. J. da Silva, M. V. Linhares, R. Padilha, N. Roqueiro, and R. S. de Oliveira, "An empirical study of sysml in the modeling of embedded systems", in *International Conference on Systems, Man and Cybernetics*, IEEE, volume 6, 2006, pages 4569–4574 (cited on page 12).

[345] E. Silva, E. P. Freitas, F. R. Wagner, F. C. Carvalho, and C. E. Pereira, "Java framework for distributed real-time embedded systems", in *Ninth International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, IEEE, 2006, 8–pp (cited on page 57).

[346] W. Smeenk, J. Moerman, F. Vaandrager, and D. N. Jansen, "Applying automata learning to embedded control software", in *International Conference on Formal Engineering Methods*, Springer, 2015, pages 67–83 (cited on pages 5, 116, 119, 130).

[347] R. Smetsers, J. Moerman, M. Janssen, and S. Verwer, "Complementing Model Learning with Mutation-Based Fuzzing", *arXiv preprint arXiv:1611.02429*, 2018 (cited on pages 115, 130).

[348] R. Smetsers, M. Volpato, F. Vaandrager, and S. Verwer, "Bigger is not always better: On the quality of hypotheses in active automata learning", in *International Conference on Grammatical Inference*, 2014, pages 167–181 (cited on page 130).

[349] W. Song, X. Xia, H. A. Jacobsen, P. Zhang, and H. Hu, "Efficient Alignment Between Event Logs and Process Models", *IEEE Transactions on Services Computing*, volume 10, pages 136–149, 2017 (cited on page 79).

[350] *Splunk*, `http://www.splunk.com`. Accessed: September 2022, 2005 (cited on pages 60, 175).

[351] D. P. Spooner and D. J. Kerbyson, "Performance feature identification by comparative trace analysis", *Future Generation Computer Systems*, volume 22, pages 369–380, 2006 (cited on pages 79, 81, 86, 87).

[352] *Springer link online library*, `https://link.springer.com/`. Accessed: April 2021 (cited on page 69).

[353] M. Staron, "Adopting model driven software development in industry–a case study at two companies", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2006, pages 57–72 (cited on page 159).

[354] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008 (cited on page 141).

[355] M. Stephan and J. R. Cordy, "Identifying instances of model design patterns and antipatterns using model clone detection", in *7th International Workshop on Modeling in Software Engineering*, IEEE, 2015, pages 48–53 (cited on page 165).

[356] M. Stephan and E. J. Rapos, "Model clone detection and its role in emergent model pattern mining", *Model Management and Analytics for Large Scale Systems*, page 37, 2019 (cited on page 165).

[357] A. Stevenson and J. R. Cordy, "A survey of grammatical inference in software engineering", *Science of Computer Programming*, volume 96, pages 444–459, 2014 (cited on page 112).

[358] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research: A critical review and guidelines", in *38th International Conference on Software Engineering*, IEEE, 2016, pages 120–131 (cited on page 139).

[359] M.-A. Storey, N. A. Ernst, C. Williams, and E. Kalliamvakou, "The who, what, how of software engineering research: A socio-technical framework", *Empirical Software Engineering*, volume 25, pages 4097–4129, 2020 (cited on pages 61, 78, 91, 102, 103, 176).

[360] P. E. Strandberg, E. P. Enoiu, W. Afzal, D. Sundmark, and R. Feldt, "Information flow in software testing–an interview study with embedded software engineering practitioners", *IEEE Access*, volume 7, pages 46 434–46 453, 2019 (cited on pages 2, 12).

[361] A. Strauss and J. M. Corbin, *Grounded theory in practice*. Sage, 1997 (cited on page 16).

[362] N. Suguna and R. M. Chandrasekaran, "Identifying the Behavioral Difference using Differential Slicing", *International Journal of Applied Information Systems*, volume 5, pages 41–48, 2013 (cited on pages 79, 86, 87).

[363] W. N. Sumner and X. Zhang, "Algorithms for automatically computing the causal paths of failures", in *12th International Conference on Fundamental Approaches to Software Engineering*, Springer, 2009, pages 355–369 (cited on pages 79, 84, 86, 87).

[364] S. Suriadi, R. S. Mans, M. T. Wynn, A. Partington, and J. Karnon, "Measuring patient flow variations: A cross-organisational process mining approach", in *Asia Pacific Business Process Management*, 2014 (cited on page 79).

[365] J. Svacina, J. Raffety, C. Woodahl, B. Stone, T. Cerny, M. Bures, D. Shin, K. Frajtak, and P. Tisnovsky, "On vulnerability and security log analysis: A systematic literature review on recent trends", in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*, 2020, pages 175–180 (cited on page 105).

[366] A. Syamsiyah, A. Bolt, L. Cheng, B. F. A. Hompes, R. P. J. C. Bose, B. F. van Dongen, and W. M. van der Aalst, "Business process comparison: A methodology and case study", in *Business Information Systems*, 2017 (cited on page 79).

[367] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads", *Automated Software Engineering*, volume 24, pages 189–231, 2017 (cited on pages 79, 83, 86, 89, 90).

[368] M. Szpyrka, E. Brzychczy, A. Napieraj, J. Korski, and G. J. Nalepa, "Conformance checking of a longwall shearer operation based on low-level events", *Energies*, volume 13, pages 1–18, 2020 (cited on page 79).

[369] B. C. Tak, S. Tao, L. Yang, C. Zhu, and Y. Ruan, "Logan: Problem diagnosis in the cloud using log-based reference models", in *International Conference on Cloud Engineering*, IEEE, 2016, pages 62–67 (cited on pages 79, 83, 86, 90, 94–96, 104).

[370] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Salsa: Analyzing logs as state machines.", *WASL*, volume 8, pages 6–6, 2008 (cited on page 3).

[371] L. Tan and C. Bockisch, "A survey of refactoring detection tools", in *Software Engineering*, 2019 (cited on page 57).

[372] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes? an exploratory study in industry", in *Proceedings of the ACM SIGSOFT 20th International symposium on the foundations of software engineering*, 2012, pages 1–11 (cited on page 105).

[373] F. Taymouri and J. Carmona, "An evolutionary technique to approximate multiple optimal alignments", in *International Conference on Business Process Management*, 2018 (cited on page 79).

[374] F. Taymouri, M. L. Rosa, and J. Carmona, "Business process variant analysis based on mutual fingerprints of event logs", *Advanced Information Systems Engineering*, volume 12127, pages 299 –318, 2019 (cited on page 79).

[375] D. Thakkar, Z. M. Jiang, A. Hassan, G. Hamann, and P. Flora, "Retrieving relevant reports from a customer engagement repository", *IEEE International Conference on Software Maintenance*, pages 117–126, 2008 (cited on pages 79, 82, 83, 86).

[376] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models", *Empirical Software Engineering*, volume 19, pages 182–212, 2014 (cited on page 142).

[377] J.-P. Tolvanen and S. Kelly, "Integrating models with domain-specific modeling languages", in *Proceedings of the 10th Workshop on Domain-specific Modeling*, ACM, 2010, page 10 (cited on page 161).

[378] N. L. Tookkit, `https://www.nltk.org/`. Accessed: November 2019, 2014 (cited on page 142).

[379] W. Torres, M. G. van den Brand, and A. Serebrenik, "Model management tools for models of different domains: A systematic literature review", in *International Systems Conference*, IEEE, 2019, pages 1–8 (cited on page 161).

[380] B. Trakhtenbrot and Y. Barzdin, "Finite automata: Behavior and synthesis", *Journal of Symbolic Logic*, volume 42, pages 111–112, 1977 (cited on page 115).

[381] A. Tsoury, P. Soffer, and I. Reinhartz-Berger, "How well did it recover? Impact-aware conformance checking", *Computing*, volume 103, pages 3–27, 2021 (cited on page 79).

[382] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)", *IEEE Transactions on Software Engineering*, volume 43, pages 1063–1088, 2015 (cited on pages 7, 134, 164).

[383] F. Vaandrager, "Model learning", *Communications of the ACM*, volume 60, pages 86–95, 2017 (cited on pages 6, 110–112, 115, 119, 131).

[384] R. Vaarandi, "A breadth-first algorithm for mining frequent patterns from event logs", in *Intelligence in Communication Systems*, 2004 (cited on page 79).

[385] M. Valenzuela, V. Ha, and O. Etzioni, "Identifying meaningful citations", in *Workshops at the twenty-ninth AAAI conference on artificial intelligence*, 2015 (cited on page 74).

[386] A. Vallecillo, "On the industrial adoption of model driven engineering. is your company ready for mde?", *International Journal of Information Systems and Software Engineering for Big Companies*, volume 1, pages 52–68, 2015 (cited on page 5).

[387] N. R. Van Beest, M. Dumas, L. García-Bañuelos, and M. La Rosa, "Log delta analysis: Interpretable differencing of business process event logs", in *13th International Conference on Business Process Management*, Springer, 2015, pages 386–405 (cited on page 79).

[388] R. Van Der Straeten, T. Mens, and S. Van Baelen, "Challenges in model-driven software engineering", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2008, pages 35–47 (cited on page 159).

[389] S. K. Vanden Broucke, J. De Weerdt, J. Vanthienen, and B. Baesens, "Determining process model precision and generalization with weighted artificial negative events", *IEEE Transactions on Knowledge and Data Engineering*, volume 26, pages 1877–1889, 2014 (cited on page 79).

[390] Verum, `http://www.verum.com`. Accessed: May 2018, 2014 (cited on pages 116, 136, 138, 159).

[391] S. Verwer and C. A. Hammerschmidt, "Flexfringe: A Passive Automaton Learning Package", in *IEEE International Conference on Software Maintenance and Evolution*, 2017, pages 638–642 (cited on page 123).

[392] J. R. M. Viana, N. P. Viana, F. A. M. Trinta, and W. V. De Carvalho, "A systematic review on software engineering in pervasive games development", in *Brazilian Symposium on Computer Games and Digital Entertainment*, IEEE, 2014, pages 51–60 (cited on page 73).

[393] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions", *Journal of Systems and Software*, volume 110, pages 54–84, 2015 (cited on pages 1, 2).

[394] N. Walkinshaw and K. Bogdanov, "Inferring finite-state models with temporal constraints", in *Proceedings of the 23rd International Conference on Automated Software Engineering*, IEEE Computer Society, 2008, pages 248–257 (cited on pages 6, 111).

[395] N. Walkinshaw, R. Taylor, and J. Derrick, "Inferring extended finite state machine models from software executions", *Empirical Software Engineering*, volume 21, pages 811–853, 2016 (cited on pages 58, 110, 112, 130).

[396] N. Walkinshaw and K. Bogdanov, "Automated comparison of state-based software models in terms of their language and structure", *ACM Transactions on Software Engineering and Methodology*, volume 22, 2013 (cited on pages 79, 81, 82, 86–88, 90, 101, 117).

[397] J. Wang, B. Cao, X. Zheng, D. Tan, and J. Fan, "Detecting Difference Between Process Models Using Edge Network", *IEEE Access*, volume 7, pages 142 916–142 925, 2019 (cited on page 79).

[398] Q. Wang and A. Orso, "Improving testing by mimicking user behavior", in *2020 International Conference on Software Maintenance and Evolution*, IEEE, 2020, pages 488–498 (cited on pages 79, 81, 86).

[399] T. Wang and A. Roychoudhury, "Automated path generation for software fault localization", *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005 (cited on pages 79, 86).

[400] T. Wang and A. Roychoudhury, "Hierarchical dynamic slicing", in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pages 228–238 (cited on pages 79, 83, 86, 91, 94, 95).

[401] T. T. Wang, K. C. Wang, X. H. Su, and Z. Lei, "Invariant based fault localization by analyzing error propagation", *Future Generation Computer Systems*, volume 94, pages 549–563, 2019 (cited on page 86).

[402] M. Weber, K. Mohror, M. Schulz, B. R. de Supinski, H. Brunst, and W. E. Nagel, "Alignment-based metrics for trace comparison", in *19th International Conference on Parallel Processing*, Springer, 2013, pages 29–40 (cited on pages 79, 81, 86, 94, 95, 104).

[403] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan, "Analyzing concurrency bugs using dual slicing", in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pages 253–264 (cited on pages 79, 81, 84, 86, 91, 94, 96, 97, 102).

[404] M. Weidlich, A. Polyvyanyy, N. Desai, and J. Mendling, "Process compliance measurement based on behavioural profiles", in *22nd International Conference on Advanced Information Systems Engineering*, Springer, 2010, pages 499–514 (cited on page 79).

[405] M. Weidlich, A. Polyvyanyy, N. Desai, J. Mendling, and M. Weske, "Process compliance analysis based on behavioural profiles", *Information Systems*, volume 36, pages 1009–1025, 2011 (cited on page 79).

[406] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik, "Process discovery using integer linear programming", in *Fundamenta Informaticae*, 2009 (cited on pages 58, 110, 162).

[407] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering", *IEEE software*, volume 31, pages 79–85, 2013 (cited on pages 6, 134, 163, 164).

[408] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "Industrial adoption of model-driven engineering: Are the tools really the problem?", in *International Conference on Model Driven Engineering Languages and Systems*, Springer, 2013, pages 1–17 (cited on pages 161, 163).

[409] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "A taxonomy of tool-related issues affecting the adoption of model-driven engineering", *Software & Systems Modeling*, volume 16, pages 313–331, 2017 (cited on page 5).

[410] R. Wieman, M. F. Aniche, W. Lobbezoo, S. Verwer, and A. van Deursen, "An experience report on applying passive learning in a large-scale payment company", in *IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2017, pages 564–573 (cited on page 162).

[411] A. Wiese, V. Ho, and E. Hill, "A comparison of stemmers on source code identifiers for software search", in *27th IEEE International Conference on Software Maintenance*, IEEE, 2011, pages 496–499 (cited on page 142).

[412] *Wiley online library*, `https://onlinelibrary.wiley.com/`. Accessed: April 2021 (cited on page 69).

[413] J. Witschey, S. Xiao, and E. Murphy-Hill, "Technical and personal factors influencing developers' adoption of security tools", in *Proceedings of the 2014 ACM Workshop on Security Information Workers*, 2014, pages 23–26 (cited on page 175).

[414] L. Wittgenstein, *Philosophical investigations*. John Wiley & Sons, 2009 (cited on page 139).

[415] C. Wohlin, E. Mendes, K. R. Felizardo, and M. Kalinowski, "Guidelines for the search strategy to update systematic literature reviews in software engineering", *Information and software technology*, volume 127, page 106 366, 2020 (cited on page 75).

[416] M. T. Wynn, E. Poppe, J. Xu, A. H. ter Hofstede, R. Brown, A. Pini, and W. M. van der Aalst, "ProcessProfiler3D: A visualisation framework for log-based process performance comparison", *Decision Support Systems*, volume 100, pages 93–108, 2017 (cited on page 79).

[417] X. Xie, Z. Jin, J. Wang, L. Yang, Y. Lu, and T. Li, "Confidence guided anomaly detection model for anti-concept drift in dynamic logs", *Journal of Network and Computer Applications*, volume 162, pages 1–10, 2020 (cited on pages 79, 86).

[418] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing", *ACM SIGPLAN Notices*, volume 43, pages 238–248, 2008 (cited on page 87).

[419] J. Xing, B. D. Theelen, R. Langerak, J. van de Pol, J. Tretmans, and J. P. Voeten, "From poosl to uppaal: Transformation and quantitative analysis", in *10th International Conference on Application of Concurrency to System Design*, IEEE, 2010, pages 47–56 (cited on page 162).

[420] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair", in *Proceedings of the 40th international conference on software engineering*, 2018, pages 789–799 (cited on pages 79, 82, 86).

[421] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs", in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pages 117–132 (cited on page 3).

[422] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs", *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020 (cited on page 79).

[423] Z. Yan, R. Dijkman, and P. Grefen, *Fast business process similarity search*. 2012, volume 30, pages 105–144 (cited on page 79).

[424] N. Yang, K. Aslam, R. Schiffelers, L. Lensink, D. Hendriks, L. Cleophas, and A. Serebrenik, "Improving model inference in industry by combining active and passive learning", in *26th International Conference on Software Analysis, Evolution and Reengineering*, IEEE, 2019, pages 253–263 (cited on pages 8, 58, 72).

[425] N. Yang, P. Cuijpers, D. Hendriks, R. Schiffelers, J. Lukkien, and A. Serebrenik, "An interview study about the use of logs in embedded software engineering", *Empirical Software Engineering*, 2022 (cited on page 8).

[426] N. Yang, P. Cuijpers, R. Schiffelers, J. Lukkien, and A. Serebrenik, "Painting flowers: Reasons for using single-state state machines in model-driven engineering", in *17th International Conference on Mining Software Repositories*, 2020 (cited on page 9).

[427] N. Yang, P. Cuijpers, R. Schiffelers, J. Lukkien, and A. Serebrenik, "An interview study of how developers use execution logs in embedded software engineering", in *43rd International Conference on Software engineering*, 2021 (cited on pages 4, 8, 76, 78, 100).

[428] N. Yang, P. Cuijpers, R. Schiffelers, J. Lukkien, and A. Serebrenik, "Single-state-state-machines in model-driven software engineering: An exploratory study", *Empirical Software Engineering*, 2021 (cited on page 9).

[429] N. Yang, D. Hendriks, J. Lukkien, and A. Serebrenik, "A literature review of log comparison techniques for software engineering", *Under review of ACM Transactions on Software Engineering and Methodology*, 2022 (cited on page 8).

[430] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge", in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pages 689–699 (cited on page 59).

[431] R. K. Yin, "Case study research: Design and methods, applied social research", *Methods series*, volume 5, 1994 (cited on page 138).

[432] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software", in *34th International Conference on Software Engineering*, IEEE, 2012, pages 102–112 (cited on pages 48, 53, 54, 59).

[433] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement", *ACM Transactions on Computer Systems*, volume 30, pages 1–28, 2012 (cited on page 3).

[434] A. Zaidman and S. Demeyer, "Managing trace data volume through a heuristical clustering process based on event execution frequency", in *CSMR*, 2004, pages 329–338 (cited on page 58).

[435] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining", *Empirical Software Engineering*, volume 16, pages 325–364, 2011 (cited on page 59).

[436] Y. Zeng, J. Chen, W. Shang, and T.-H. P. Chen, "Studying the characteristics of logging practices in mobile apps: A case study on f-droid", *Empirical Software Engineering*, volume 24, pages 3394–3434, 2019 (cited on pages 19, 48–50, 54).

[437] H. Zha, J. Wang, L. Wen, C. Wang, and J. Sun, "A workflow net similarity measure based on transition adjacency relations", *Computers in Industry*, volume 61, pages 463–471, 2010 (cited on page 79).

[438] C. Zhang and D. Budgen, "What do we know about the effectiveness of software design patterns?", *IEEE Transactions on Software Engineering*, volume 38, pages 1213–1231, 2011 (cited on page 162).

[439] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie, "Software analytics in practice", *IEEE software*, volume 30, pages 30–37, 2013 (cited on pages 4, 168).

[440] L. Zhang and C. Wang, "Rclassify: Classifying race conditions in web applications via deterministic replay", *IEEE/ACM 39th International Conference on Software Engineering*, pages 278–288, 2017 (cited on pages 79, 81, 86, 90, 94, 96, 97, 102, 103).

[441] X. Zhang and R. Gupta, "Matching execution histories of program versions", *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 197–206, 2005 (cited on pages 79, 86, 87).

[442] L. Zhao, Z. Zhang, L. Wang, and X. Yin, "A fault localization framework to alleviate the impact of execution similarity", *International Journal of Software Engineering and Knowledge Engineering*, volume 23, pages 963–998, 2013 (cited on pages 67, 86).

[443] C. Zheng, L. Wen, and J. Wang, "Detecting process concept drifts from event logs", in *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE*, Springer, 2017, pages 524–542 (cited on page 79).

[444] P. Zhou, B. Gill, W. Belluomini, and A. Wildani, "Gaul: Gestalt analysis of unstructured logs for diagnosing recurring problems in large enterprise storage systems", in *29th Symposium on Reliable Distributed Systems*, IEEE, 2010, pages 148–159 (cited on pages 79, 84, 86).

[445] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study", *IEEE Transactions on Software Engineering*, volume 47, pages 243–260, 2021 (cited on pages 66, 68, 79).

[446] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions", in *37th IEEE International Conference on Software Engineering*, IEEE, volume 1, 2015, pages 415–425 (cited on pages 26, 54).

[447] S. Zhu, X. Hu, Z. Qian, Z. Shafiq, and H. Yin, "Measuring and disrupting anti-adblockers using differential execution analysis", in *The Network and Distributed System Security Symposium*, 2018 (cited on page 79).

[448] *Zipkin 2022*, `https://zipkin.io/`. Accessed: September 2022 (cited on page 103).

[449] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, "Mimic: Locating and understanding bugs by analyzing mimicked executions", in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pages 815–826 (cited on pages 79, 86, 90, 94, 98, 102).

[450] syntok, `https://github.com/fnl/syntok`. Accessed: September 2022, 2014 (cited on page 142).

Summary

Embedded production systems are nowadays widely used in production and manufacturing for a more precise control of the production processes and the quality of products. Such systems are hard to maintain due to their complex nature. They consist of thousands of software and hardware components that are communicating with each other, capture the concepts and designs related to different disciplines (e.g., physics and chemistry), and are often featured with critical performance requirements. The portion of system functionality realized by software is increasing in such systems, rendering it increasingly complex. According to the existing empirical studies, the general software engineering methodologies and techniques might not be sufficient to address the challenges raised by the properties of embedded production software, suggesting the need for proposing tailored techniques for such systems. As the first step towards proposing effective techniques, it is essential to understand the current practices and the challenges developers are facing. To gain this understanding, we conduct a series of empirical studies at ASML, a leading manufacturer of lithography machines for semiconductor industry.

Similar to many other complex systems, systems at ASML generate a large amount of logs that capture the runtime behavior of the systems. Due to the presence of rich information, execution logs are considered to be essential inputs for software analytics tools and processes that aim at addressing the complexity of large-scale systems. To provide useful and effective software analytics tools for complex embedded systems, it is hence necessary to understand how the developers of such systems use logs in practice. We conduct an interview study with 39 software developers (Chapter 2). We first perform a series of interviews with 25 developers at ASML and then replicate the interviews at four other companies with another 14 software developers. In this interview study, we learn that developers often compare logs generated from multiple executions to support their maintenance activities such as root cause analysis and behavioral verification. While many log comparison techniques have been proposed in the academic literature, text-based editors are the commonly used tools for this practice. This observation leads us to study the existing log comparison techniques and their limitations by conducting a literature review about the existing log comparison techniques (Chapter 3). This literature study reveals that most of the existing log comparison techniques do not explicitly take the industrial challenges (discussed in Chapter 2) into account and were evaluated in a limited way without involvement of human participants. To provide software developers with effective log comparison techniques, we suggest

researchers to improve log comparison techniques to address the industrial challenges, and evaluate the techniques in a natural development setting.

In the interview study about log analysis practice (Chapter 2), we also observe that developers often manually sketch behavioral models based on logs. This preference of presenting log information with models co-occurs with the transition ASML is taking from code-based software engineering to model-driven software engineering (MDSE). To enable the use of MDSE, ASML needs to create models for the existing code-based components. To facilitate the automation of model creation, we propose a model inference technique that can extract models by combining log analysis, and analysis of a running system under stimuli (Chapter 4). The proposed technique significantly outperforms the existing techniques, as evaluated with 18 ASML software components. However, there are still many theoretical and practical challenges to be addressed in order to apply model inference techniques in industry. Due to these challenges, models are manually created by software developers in practice. We therefore turn our attention to study how developers manually create models for MDSE. Particularly, we study why developers violate modeling guidelines which are considered as common wisdom. We focus on an extreme case, known as flower models, consisting of only a single state. Combining qualitative and quantitative analyses, we identify the main reasons of the guideline violations, providing empirical evidence on the challenges in MDSE, and suggestions on the improvements of MDSE tools and guidelines (Chapter 5).

In summary, this thesis presents a series of empirical studies conducted in industry, which provides an overview of challenges faced by developers when using logs and models in the context of a transition from code-based engineering to MDSE. The empirical evidence collected from these studies supports researchers and tool builders to develop techniques for facilitating the transition to MDSE for embedded production systems.

Curriculum Vitae

Nan Yang was born in Youxi, China, and completed her bachelor's degree in Electrical Engineering at Shanghai Maritime University. She then pursued a master's degree in Embedded Systems at Eindhoven University of Technology, where she was supervised by Prof. Alexander Serebrenik.

In 2018, Nan began her Ph.D. candidacy at the Eindhoven University of Technology, working in the group Interconnected Resource-aware Intelligent Systems. Her research project was a collaboration between ASML and Eindhoven University of Technology, where she focused on the challenges that developers face when using logs and models in the context of transitioning from code-based engineering to model-driven engineering for embedded production systems. The scientific contributions are presented in this thesis. Besides, Nan has also worked on identifying the motivations of OSS projects joining the Apache foundation.

Following the completion of her Ph.D., Nan will take on the role of Researcher at TNO-ESI, where she will continue to investigate methods for dealing with the complexity of high-tech systems.

Titles in the IPA Dissertation Series since 2020

M.A. Cano Grijalba. *Session-Based Concurrency: Between Operational and Declarative Views*. Faculty of Science and Engineering, RUG. 2020-01

T.C. Nägele. *CoHLA: Rapid Co-simulation Construction*. Faculty of Science, Mathematics and Computer Science, RU. 2020-02

R.A. van Rozen. *Languages of Games and Play: Automating Game Design & Enabling Live Programming*. Faculty of Science, UvA. 2020-03

B. Changizi. *Constraint-Based Analysis of Business Process Models*. Faculty of Mathematics and Natural Sciences, UL. 2020-04

N. Naus. *Assisting End Users in Workflow Systems*. Faculty of Science, UU. 2020-05

J.J.H.M. Wulms. *Stability of Geometric Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2020-06

T.S. Neele. *Reductions for Parity Games and Model Checking*. Faculty of Mathematics and Computer Science, TU/e. 2020-07

P. van den Bos. *Coverage and Games in Model-Based Testing*. Faculty of Science, RU. 2020-08

M.F.M. Sondag. *Algorithms for Coherent Rectangular Visualizations*. Faculty of Mathematics and Computer Science, TU/e. 2020-09

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic*. Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components*. Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management*. Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS*. Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning*. Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs*. Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical*. Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions*. Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition*. Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Concurrent Separation Logics for Safety, Refinement, and Security*. Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing*. Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving*. Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution*. Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and Models in Engineering Complex Embedded Production Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2023-03