An interview study about the use of logs in embedded software engineering[*]

Nan Yang · Pieter Cuijpers · Dennis
Hendriks · Ramon Schiffelers · Johan
Lukkien · Alexander Serebrenik

Abstract *Context:* 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 embedded software engineering.

Objective: In this study, we aim to understand how developers use logs in an embedded software engineering context. Specifically, we would like to gain insights into the type of logs developers analyze, the purposes for which developers analyze logs, the information developers need from logs and their expectation on tool support.

---

[*] This paper extends a published paper [93]

Nan Yang
Eindhoven University of Technology, The Netherlands
E-mail: n.yang1@tue.nl

Pieter Cuijpers
Eindhoven University of Technology, The Netherlands
Aalborg University, Denmark
E-mail: p.j.l.cuijpers@tue.nl

Dennis Hendriks
ESI (TNO), The Netherlands
Radboud University, The Netherlands
E-mail: dennis.hendriks@tno.nl, dennis.hendriks@ru.nl

Ramon Schiffelers
ASML, The Netherlands
Eindhoven University of Technology, The Netherlands
E-mail: r.r.h.schiffelers@tue.nl

Johan Lukkien
Eindhoven University of Technology, The Netherlands
E-mail: j.j.lukkien@tue.nl

Alexander Serebrenik
Eindhoven University of Technology, The Netherlands
E-mail: a.serebrenik@tue.nl

Method: In order to achieve the aim, we conducted these interview studies. First, we interviewed 25 software developers from ASML, which is a leading company in developing lithography machines. This exploratory case study provides the preliminary findings. Next, we validated and refined our findings by conducting a replication study. We involved 14 interviewees from four companies who have different software engineering roles in their daily work.

Results: As the result of our first study, we compile a preliminary taxonomy which consists of four types of logs used by developers in practice, 18 purposes of using logs, 13 types of information developers search in logs, 13 challenges faced by developers in log analysis and three suggestions for tool support provided by developers. This taxonomy is refined in the replication study with three additional purposes, one additional information need, four additional challenges and three additional suggestions of tool support. In addition, with these two studies, we observed that text-based editors and self-made scripts are commonly used when it comes to tooling in log analysis practice. As indicated by the interviewees, the development of automatic analysis tools is hindered by the quality of the logs, which further suggests several challenges in log instrumentation and management.

Conclusions: Based on our study, we provide suggestions for practitioners on logging practices. We provide implications for tool builders on how to further improve tools based on existing techniques. Finally, we suggest some research directions and studies for researchers to further study software logging.

Keywords log analysis practice, embedded software enigineering

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 [78], test generation [73, 21], and change comprehension [4, 10, 63]. However, researchers have provided empirical evidence that log analysis tools are not necessarily effective and applicable when dealing with real-world problems [64]. Legunsen et al. [52] 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-art log analysis tools failed in processing the large volume of logs produced by a large-scale embedded system [64].

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. [56] 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. [11] 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 [35]. 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 [86], modeling [83], and architecture design [5], 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 (RQ3) and what tool support could be useful (RQ4). 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) and the purposes for which developers analyze logs (RQ2).

To answer our research questions, in our previous conference paper [93] we reported on an exploratory case study at ASML, a company that develops lithography systems for the semiconductor industry. We conducted a series of semi-structured interviews with 25 software developers. We observed that developers use four types of execution logs that record high-level machine actions and errors, low-level execution details, performance data as well as business-critical data (RQ1). We confirmed that logs are primarily used for analyzing software issues [11, 56]. In addition, we observed the use of logs, e.g., for test code development and requirement reverse-engineering (RQ2). We identified 13 types of information developers search for in the execution logs. We found that the most frequently mentioned types of information are *propagation of errors across systems*, *timestamps associated with log lines*, *data flow*, *interaction of software components*, and *differences between multiple executions* (RQ3). In addition to the common challenges related to log quality [99, 56], we observed that the *lack of domain knowledge*, *lack of familiarity with code base and software design* and *presence of concurrency* raise major challenges in log analysis for such complex and multidisciplinary systems. Particularly, developers shared that obtaining a high-level picture of component interactions is useful for developing global comprehension on the behavior of such systems. Such abstraction is particularly hard to obtain with currently used tools (e.g., text-based tools). Thus, developers expect tools to help them handle the complexity by adding multi-level abstractions to logs and comparing multiple logs on different levels of abstraction (RQ4).

The exploratory case study 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, in the current work we extend our previous 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 (RQ5). 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 (RQ5). 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 paper is organized as follows. In Section 2, we present our exploratory study at ASML. Next, in Section 3, we report on our replication study at four companies. Based on these two studies, we synthesize the findings in Section 4. We then discuss the implication of our work in Section 5. We discuss threats to validity in Section 6. Finally, we conclude in Section 7.

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 2.1). The results of this study that answer our research questions are then presented in Sections 2.2 (RQ1), 2.3 (RQ2), 2.4 (RQ3) and 2.5 (RQ4).

2.1 Methodology

To understand how software developers use logs in the embedded systems industry, we conducted a case study [77]. As our research questions differ from previous work [11, 56], we opted for an exploratory rather than a confirmatory study.

2.1.1 Context

Our study is part of an ongoing collaboration with ASML which 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 [50].

The software components developed by this division form a paradigmatic subsystem [29] 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 interprocess 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 [7], 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 [16]. 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.

2.1.2 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 [15]. Table 1 shows our interview guide. In adherence to best interviewing practices [15], 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.

Table 1: 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)
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 (RQ2)
6. For what purposes do you use them?
7. How often do you analyze logs for your purposes?
Information needs (RQ3)
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 (RQ4)
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?

2.1.3 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). 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.

2.1.4 Data collection and analysis

We collected data by recording the audio and making the transcripts. We coded the transcripts [15] using the *ATLAS.ti* data analysis software. Our coding process consists of three steps. First, we performed open coding. We

Table 2: Background of interviewees

Group ID	Participant ID	Years of experience	Current role[1]	Gender[2]	Education background[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.

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 [87], 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 particiapnt.

Table 3: Four types of logs (RQ1)

Type	Information	Enabled by default	Presence of physical layers	Quote ID
Event log (EL)	Machine event & error message	✓	Not necessary	P20-1
Function trace (FT)	Order of functions & values of parameters	✗	Not necessary	P18-1
Performance data (PD)	Duration of software & hardware actions	✓	Necessary	P7-1
Functional data (FD)	Business-critical data	✓	Not necessary	P8-1

2.1.5 Member checking

Coding is an interpretative process and as such there is always a risk of misinterpretation [43]. In order to reduce this risk, we performed member checking [17], 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 Type of logs (RQ1)

The types of execution logs are summarized in Table 3.

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 summary:** Developers use different types of execution logs that record high-level machine actions, low-level execution details, throughput information as well as bushiness-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.

2.3 Purpose of log analysis (RQ2)

We identified 18 purposes and classified them into four categories. Our findings are complementary to the prior studies [56, 11, 98]. Consistent with the prior studies, we found that developers primarily use logs for the purposes of analyzing issues. We also identified purposes (e.g., developing test scenario & code, reverse-engineering requirements for legacy software and identifying root cause of flaky executions) not previously discussed in the literature.

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).

Table 4: 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

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).

2.3.3 Verification and improvement

Execution logs help developers to verify and optimize different aspects of the software. In addition to running tests against requirements, developers extensively inspect logs to verify whether the software behaves as expected: "*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).

2.3.4 *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 [62]:"*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).

2.3.5 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 4 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 debugging approaches 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 time-out, 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 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).

RQ2 summary: Developers rely on logs to obtain low-level execution information for issue analysis that cannot be easily obtained using traditional debugging approaches. Our findings complement the literature and provide empirical evidences for some additional purposes (e.g., test development).

Table 5: 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 from different executions differ?	12	P3-5
How do the functional data from different executions differ?	7	P7-5

2.4 Information needs (RQ3)

We grouped information needs into five categories as shown in Table 5. We observed that developers tend to have common information needs; five types of information are mentioned by more than 10 developers ($> 40\%$ interviewees).

2.4.1 *Context of issues*

As discussed in the previous section, developers use logs for issue analysis. Many of these issue analysis activities (e.g., identifying responsibilities) require developers to get the context of the issues: "*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 [70]. 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 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).

2.4.2 *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).

2.4.3 *Software state and interaction*

To understand the software system, developers analyze the interactions between software 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).

2.4.4 *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).

2.4.5 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 [62]:*"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 function calls 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.

> **RQ3 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.

2.5 Tool support for log analysis (RQ4)

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.

2.5.1 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).

2.5.2 Challenges in log analysis

Table 6 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, 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 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 [30, 55, 99]. 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 struc-*

tures 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 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

Many challenges are related to complexity of the system: presence of multiple interacting components, multidisciplinary context and concurrency.

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 2.4.3), 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 realised by the underlying concurrent software: *"all those process elements they end up in different lists, and then the lists are emptied by different subprocesses... and they all do their things separately and they synchronize on 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 sequentially, i.e., developers should disentangle interleaved executions.

Complexity does not only hinder comprehension but also introduce 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 pre-processing 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).

2.5.3 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*

Table 6: 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	P17-1,P11-1,P11-2,P15-3,P17-2,P17-3
Expertise		
Lack of domain knowledge	10	P11-3,P22-5,P22-6
Unfamiliar with code base and software design	9	P7-6,P15-4

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 2.4.3, 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).

RQ4 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.

---

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 (RQ5). We start with our methodology (Section 3.1) and then report our findings (Section 3.2).

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. [82] 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 [82]. In this study, we opted for dependent replications by changing the study context while following the same research method (i.e., interviews).
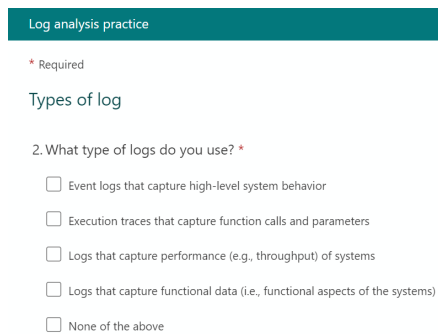
Next, we introduce the design of our interviews, context and participants as well as data collection and analysis.

3.1.1 Semi-structured interviews

We used the same research method adopted in our previous study at ASML (Section 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 1).

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 3–6 and codes discussed in Section 2.5.3) with a survey-like form: Figure 1 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 2.5.3) especially if their current tools are primitive. Therefore, we dropped the closed questions related to tool suggestions (codes presented in Section 2.5.3). In this study, 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.

Fig. 1: Closed question about type of logs used in practice

Table 7: 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

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 developers from each company. This replication study is different from our previous in-depth study of a single company (Section 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 7 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 consumer electronic[6]. Two engineers were recruited. One of them is responsible for a 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.

3.1.3 Data collection and analysis

To collect and analyze the data, we applied the same method as our study at ASML (Section 2.1.4). 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 [79]. 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.

---

[6] Company B and C are developing different kinds of consumer electronics.

3.2 Generalizability (RQ5)

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), information needs (RQ2), challenges (RQ3) and tool support (RQ4).

3.2.1 Types of logs

As identified in ASML (Section 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.

RQ5-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.

3.2.2 Purpose of log analysis

Figure 2 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 3 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 we have observed in ASML (Section 2.3.5), logs are heavily used for issue analysis for retrieving execution details because the traditional debugger is often not applicable due to the synchronization errors introduced by breakpoints. Our replication study confirms that logs are indeed essential in issue analysis for embedded systems where timing requirements are critical: "*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).

We do not observe these additional purposes at company D. Therefore, we conjecture that using logs for liability analysis, use case analysis and testing might be specific to embedded software companies where the machines are composed of many hardware components, interacting with the machine operators and performing actions based on the state of the machines.

> **RQ5-b summary:** The purposes of log analysis collected from the previous study are largely shared by other ES companies. Among these purposes, problem localization and performance improvement are voted by all the participants. Consistently with the previous observation, the traditional debugger is often not applicable in ES context, emphasising the importance of logs in issue analysis. Additionally, we identified that embedded software companies use logs for liability analysis, use case analysis and testing, which were not discovered in the previous study.

3.2.3 Information needs

Figure 4 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 2.4) are greatly generalizable to other companies. These most
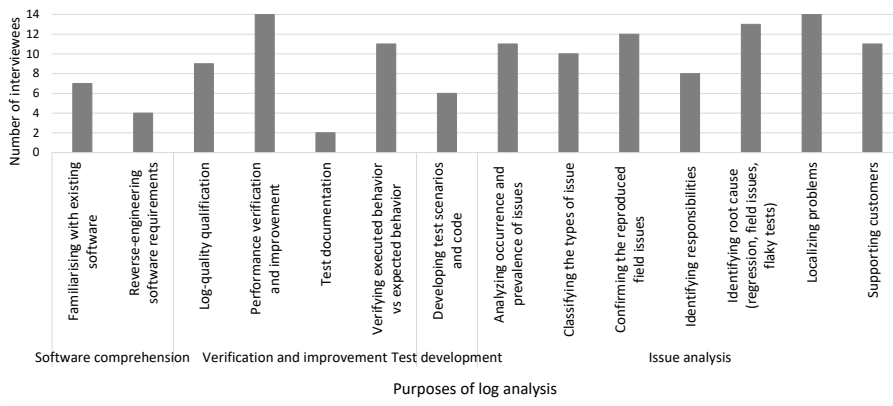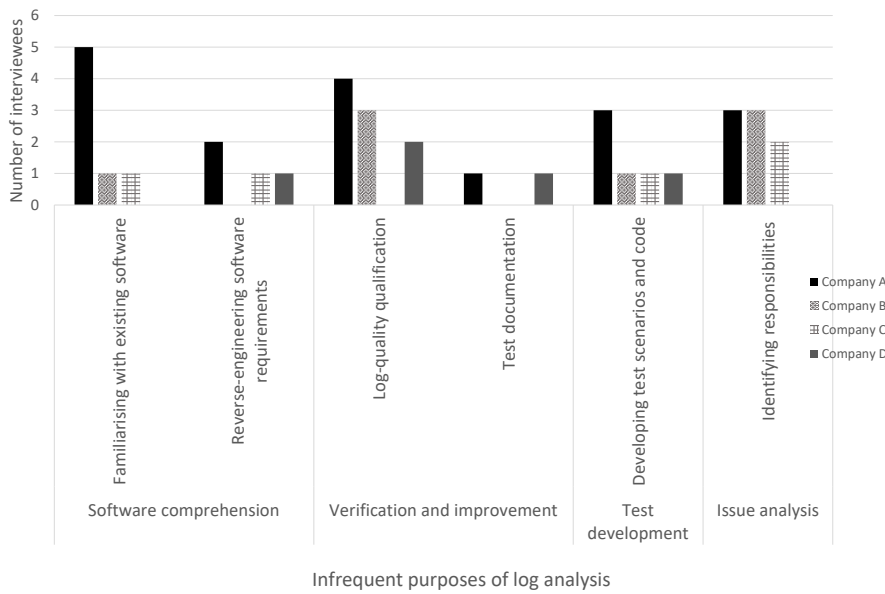
Fig. 2: Frequency of purposes in log analysis

Fig. 3: Infrequent purposes of log analysis over companies

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.
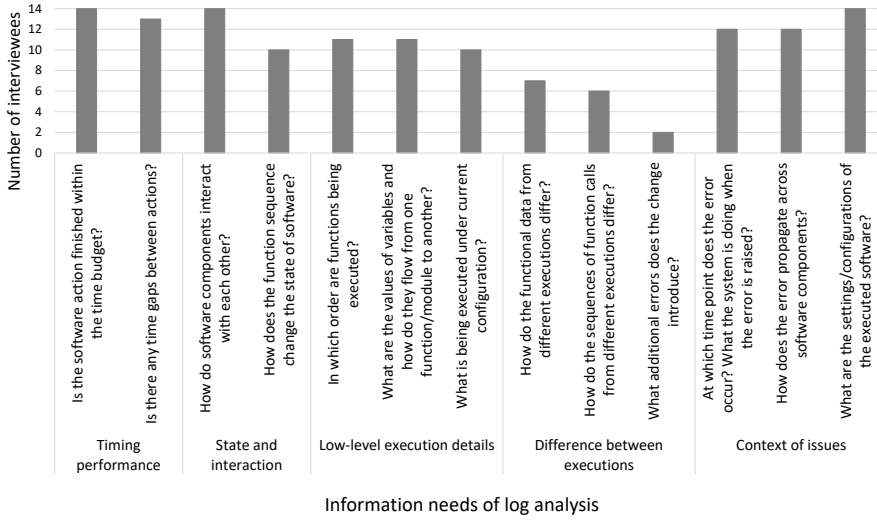
The chart shows "Number of interviewees" on the y-axis (0 to 14) with the following bars grouped by category:

Timing performance
- Is the software action finished within the time budget? — 14
- Is there any time gaps between actions? — 13

State and interaction
- How do software components interact with each other? — 14
- How does the function sequence change the state of software? — 10

Low-level execution details
- In which order are functions being executed? — 11
- What are the values of variables and how do they flow from one function/module to another? — 11
- What is being executed under current configuration? — 10

Difference between executions
- How do the functional data from different executions differ? — 7
- How do the sequences of function calls from different executions differ? — 6
- What additional errors does the change introduce? — 2

Context of issues
- At which time point does the error occur? What the system is doing when the error is raised? — 12
- How does the error propagate across software components? — 12
- What are the settings/configurations of the executed software? — 14

Information needs of log analysis

Fig. 4: 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[7] 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).

> **RQ5-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.

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, search-

---

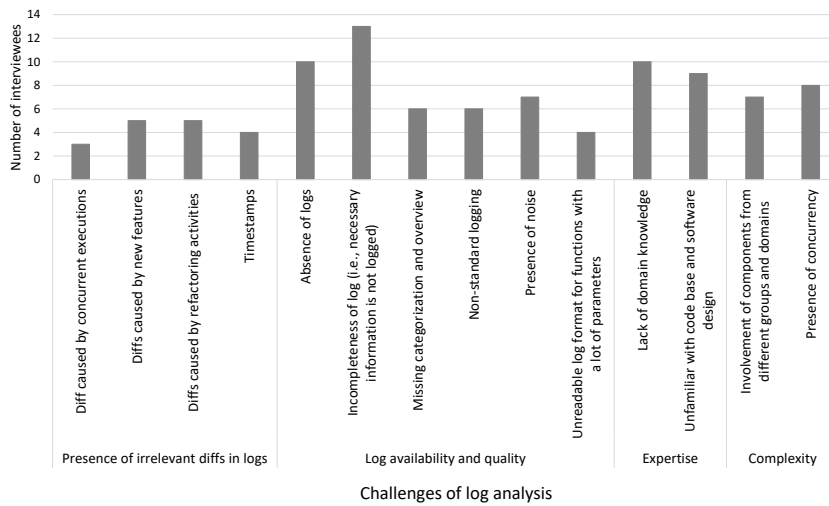[7] cycle time is the time spent on producing an item

Fig. 5: Frequency of challenges

ing, extraction and comparison, the text-based tools such as a text editor and Linux 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 or a small group of developers who work on the same part of software and share the same needs in practice: *"I made some tooling for myself where I can just visualize the interaction with the controller in sequence diagrams"* (P39-1).

RQ5-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.

3.2.5 Challenges

As shown in Figure 5, 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. 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. Furthermore, the challenges related to expertise and complexity are recognized by more than half of the interviewees (>7). Through the open discussion, 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 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 2.5.2), 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.

> **RQ5-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.

3.2.6 Expected tools

With the discussion triggered by the open questions, we identify three suggestions on tool development which were not discussed in our previous 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 ma-*

chine 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 2.5.3). 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 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 [40] 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.

RQ5-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.

---

4 Result synthesis

In this section, we synthesize the two studies presented in Sections 2 and 3. We first discuss the main scenarios of software logging identified in these two studies. As presented in Section 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.

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 2.3 and 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 interac-*

tion, 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.

Often times, developers may not have sufficient architectural and code knowledge (especially junior developers). Comparing logs generated from failing and passing executions serves as a way to identify the log information that may point to the location and root cause of issues. This comparison is often performed for different types of logs, as discussed in Section 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 comparison tools (as discussed in sections 2.5.3 and 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 2.5.2, 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 specified 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 may vary.

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.

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 which 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 debugging approach. 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 [31]. 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). 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 on performance overhead but more on identifying relevant information from a large amount of log information.

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.

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.

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).

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 [32, 40, 18]. 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 5.1). Second, we provide the refined taxonomy obtained from our work and compare the taxonomy with relevant work (Section 5.2). Third, we highlight the main findings of our work and discuss their alignment with relevant empirical work (section 5.3 and 5.4).

Finally, we discuss the recent research about log analysis techniques at ASML (Section 5.5) since the completion of our exploratory study at ASML. These research studies confirm the usefulness of our findings and implications

Table 8: Literature about logging practices. "-" indicates that the information is unspecified in the corresponding paper.

Reference	Topic	Method	Context	Domain	Language
Yuan et al. [95]	Log prevalence and modification	A mining study of four projects	OSS	Various	-
Chen et al. [19]	Log prevalence and modification	A mining study of 21 Apache projects	OSS	Various	Java, C and C++
Pecchia et al. [72]	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. [56]	Benefit and cost of logging	A survey of 66 developers and a case study of 223 logging-related issue reports.	OSS	-	-
Rong et al. [76]	Logging intentions and concerns	A series of interviews and a mining study of three projects	Industry	Big-data technology	Java
Rong et al. [75]	Consistency of logging practice	A mining study of 28 projects	OSS	Various	Java
Harty et al. [39]	Logging prevalence and information	A mining study of 57 projects	OSS	Mobile App	Java
Fu et al. [30]	Logging points	A mining study of two systems and a questionnaire survey with 54 developers	Industry	Cloud application	C#
Zeng et al. [98]	Logging purposes	A mining study of 1,444 projects and an email interview	OSS	Mobile App	Java
Barik et al. [11]	Logging purposes and challenges	Interviews with 28 software engineers, and a quantitative survey of 1,823 respondents	Industry	Cloud application	-
Kabinna et al. [46]	Impact of logging library migration	A mining study of 233 Apache projects	OSS	Various	Java
Gholamian et al. [33]	Logging overhead	An experimental study on seven Spark benchmarks	OSS	Distributed system	Java
Kabinna et al. [47]	Logging modification and stability	A mining study of four projects	OSS	Various	Java
Shang et al. [81]	Information needs of users	A qualitative analysis of 15 email inquiries and 73 inquiries from web search about different log lines	OSS	Distributed system	-

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.

5.1 Topic and context of relevant work

Table 8 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. [18] 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 (e.g., how well is the logging?) and identify the techniques that can aid developers in the log management.

Second, our study contributes to the understanding of log analysis practices for embedded systems. We can see from Table 8 that previous studies are conducted for various types of systems (e.g., cloud applications and Mobile App). As identified by Gholamian et al. [32], 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 9: Refined taxonomy for log analysis. "∗" indicates the codes that are newly discovered in the replication study but not in the exploratory 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	[72]	P20-1
Function trace	[72]	P18-1
Performance data	New	P7-1
Functional data	[72]	P8-1
Purposes	Ref./new	Quote ID
Software comprehension		
Familiarizing with existing software	[56]	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	[56, 11]	P13-1
Performance verification and improvement		
Verifying timing (throughput) performance	[98]	P16-2
Identifying opportunities of throughput improvement	[98]	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*	[11]	P35-4
Use case analysis*	[11]	P35-3
Liability analysis*	New	P34-1

Issue analysis

	Ref./new	Quote ID
Classifying the type of issues	New	P21-2
Identifying responsibilities	[56]	P4-1
Localizing problems	[98]	P1-1
Confirming reproduced field issues	New	P3-2
Identifying root cause		
Identifying root cause of field issues	[56, 76, 19, 98, 11]	P1-2
Identifying root cause of test issues	[56, 11]	P13-2
Identifying root cause of flaky (test) executions	New	P12-2
Analyzing occurrence and prevalence of issues	New	P22-1
Supporting customers	[56, 11]	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 the 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 the control sequences from different executions differ?	New	P3-5
How do the functional data from different executions differ?	New	P7-5
How do the timing behavior from different executions differ?*	New	P31-2

Challenges	**Ref./new**	**Quote ID**
Log availability and quality		
Absence of logs	[56]	P9-4,P8-2
Non-standard logging	[72, 75]	P12-4
Incompleteness of log	[56]	P9-5
Presence of noise	[56]	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	[11]	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	[37]	P11-1,P15-3,P17-3
Timing variation	[37]	P17-1
Refactoring	[37]	P11-2
New feature implementation	[37]	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	P11-3,P22-5,P22-6
Unfamiliar with code base and software design	New	P7-6,P15-4
Logging*		
Logging trade-off*	[56, 72]	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

Tool support	Ref./new	Quote ID
Creating multi-level abstraction	New	P14-5,P9-6,P17-4
Automatic log comparison	[37]	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*	[72, 76, 56]	P34-4,P34-5

5.2 Refined taxonomy for log analysis

As we can see from Table 8, we study several overlapping topics (e.g., logging purposes and challenges) with existing work in a different context. Next, we discuss the alignment in details. We revise the taxonomy for log analysis presented in Section 2 and augment it with the results of the replication study presented in Section 3. Table 9 shows a refined taxonomy. The replication study added three additional purposes related to *verification and improvement*, one information need related to *difference between executions*, two challenges related to *presence of various kinds of log differences*, three challenges related to *logging* and three suggestions on tool support.

When comparing the refined taxonomy with recent studies on logging practices, we can see that our study adds new codes to existing empirical literature of logging practices in terms of types of logs, purposes, information needs, challenges and tool support. Table 10 summarizes the main findings of our study and their alignment with existing literature. Our findings are related to the phases of both log instrumentation and log management. Next, we discuss the findings in details.

Table 10: 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 [40, 36]. 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. [76]. 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. [72], Harty et al. [39], and Shang et al. [81] 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 [11] and Google [37]. In our study, we detail the type of information developers compare and the challenges they face in practice.	Identifying the sources and causes 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.	No 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 [95, 19, 46]. 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 [71, 11].	Identifying gaps between the state-of-practice and state-of-the-art of log instrumentation and log management (R).

5.3 Log instrumentation

Among the challenges identified in Table 9, seven are related to log availability and quality. The interviewed companies largely follow the method of conventional logging [18] 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 significant amount of effort has been made in the research community to study such questions as where to log [30, 55], what to log [99], how to log [19] and how to use logs [38]; and such challenges as absence of logs [56], non-standard logging [72], and presence of noise and incomplete logging [56].

5.3.1 Logging in embedded systems

Embedded systems are comprised of various types of components, structured in different abstraction layers, and implemented with multiple programming languages [89, 35, 51]. As we discussed in Section 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. [98] 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. [33] 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 no 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 4.2). It is conjectured in the literature that OSS projects

with a different programming language may have different logging practices: Chen and Jang [19] 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. [95]. 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.

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 from interviewees is aligned with a suggestion from Rong et al. [74]. Moreover, missing logging guidelines that systematize the logging process have been reported by existing studies [74, 72]. 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 [18], 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 [69, 71]. 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 that are 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.

5.4 Log management

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

5.4.1 Multiple types of logs

As shown in Table 9, 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. [72] 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. [39] 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. [81] 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 2.4.1 and 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 [8] and change impact analysis [48].

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 [34, 4, 10, 63]. 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 [4] 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. [10] 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 of log differences can help developers perform maintenance 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 [28] and refactoring detection [88]. Chen et al. [19] 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 [12, 22, 59]. The partial ordering relation between events can be captured with logical clock timestamps [26, 66] 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.

5.4.3 Log comprehension

No previous study has investigated how developers comprehend logs. As discussed in Section 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 2.5.2), 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 [35]: 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 2.5.2) and other ES companies (Section 3.2.2) that concurrent design and time-out mechanisms, implemented in embedded systems to optimize and limit software execution time [84, 42], also hinder log comprehension. We further observed that interleaving of concurrent executions incurs challenges not only in program comprehension [6, 27] but also in log comprehension (see Section 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 [80]—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 [54]. 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 2.4).

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 [61, 90, 14, 91, 65, 13] for various software engineering activities. Beschastnikh et al. [13] designed a tool that helps developers comprehend distributed systems by visualizing the communication patterns between hosts. To help the debugging process, Mashhadi et al.[65] 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 [60, 90, 49], sets of temporal properties [53], and execution patterns [96]. 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 [92]. Moreover, these tools provide only one level of abstraction, not meeting the expectations of the interviewees (see Section 2.5.3) because the important information might be lost by showing only a certain level of detail. Several studies addressed this limitation [45, 13, 25] 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 [45, 13].

This leads us to the second implication that tool builders may take the context of use 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 [3], ranking relevant files for bug reports [94], mining requirement knowledge [58] and code summarization [67].

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 2.5.2). We expect that augmenting logs with additional knowledge derived from source code and documentation, can reduce the time and effort that developers spend in searching for information that is currently spread over multiple sources such as source code, documentation and logs.

5.4.4 The problem of co-evolution

Our work extends the discussion of the evolution of logging code [95, 19, 46]. Kabinna [47] 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 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. [95] and
Li et al. [57] 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. [57] further discovered that logging code with similar context may
need similar modifications. Therefore, the authors trained a machine learn-
ing 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 [97], metamodels and models [20, 68], and
requirements of different components [24] 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?).

5.4.5 Manual analysis with text editors

Consistently with the previous study at Microsoft [11], we found that devel-
opers mainly use text editors for their log analysis activities. Given that many
log analysis tools have been proposed over the years, 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 ad-
dress 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. [40], over the years, commercial
(e.g., Splunk [1]) and open-source tools (e.g., GrayLog [2]) 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 [18] and log abstraction [23]. 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 [85] 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 [85], 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.

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). Hooimeijer et al. [44] 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 2.5.3).

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. [41] 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 link the behavioral differences obtained from the comparison to their sources, and to provide developers with actionable insights (as discussed in Section 5.4.2).

The existence of these techniques, as well as their positive empirical validations, confirm that our findings and implications are insightful for researchers and tool builders to address log analysis challenges for embedded systems. Furthermore, they provide an example of how the research outcome of our study can be transferred to solve real-world problems.

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 first author. 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. For the first study at ASML, we have obtained 20 replies out of 25 interviewees, and the revisions requested by the interviewees were minor, suggesting high degree of validity of our interpretation. 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 on-going 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 [35]. 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 [9] 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 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.

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 for inspecting and comparing logs, despite that many log analysis tools have been proposed in literature. Our study identifies the challenges in log analysis. We observed that the unsatisfactorily log quality, lack of expertise and

high complexity of systems raise major challenges in log analysis. 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.

Based on the study, we provide suggestions for practitioners on logging practices, tool builders on how to further improve log analysis tools, and researchers on the research directions. Our observations suggest practitioners to design and implement a suitable logging process where logging approach, library, content and location as well as log generation and collection are systematized. We suggest tool builders to develop advanced log comparison tools that can categorize log differences to provide actionable insights for developers. Furthermore, our study also calls for more research efforts in supporting the evolution of log-dependent entities.

8 Data availability

The interview transcripts that support the findings of this study are not openly available due to the confidentiality agreement made with the participating companies.

9 Conflict of interest

The authors declare no conflicts of interest.

References

1. (2005) Splunk. Http://www.splunk.com
2. (2020) A leading centralized log management solution. Https://www.graylog.org
3. Aggarwal K, Timbers F, Rutgers T, Hindle A, Stroulia E, Greiner R (2017) Detecting duplicate bug reports with software engineering domain knowledge. Journal of Software: Evolution and Process 29(3):e1821
4. Amar H, Bao L, Busany N, Lo D, Maoz S (2018) Using finite-state models for log differencing. In: ESEC/FSE, pp 49–59
5. Antonino PO, Morgenstern A, Kuhn T (2016) Embedded-software architects: It's not only about the software. IEEE Software 33(6):56–62
6. Artho C, Havelund K, Honiden S (2007) Visualization of concurrent program executions. In: COMPSAC, pp 541–546
7. Asadollah SA, Inam R, Hansson H (2015) A survey on testing for cyber physical system. In: ICTSS, pp 194–207
8. Asuncion H, Asuncion A, Taylor R (2010) Software traceability with topic modeling. In: ICSE (1), pp 95–104
9. Baltes S, Ralph P (2020) Sampling in software engineering research: A critical review and guidelines. arXiv preprint arXiv:200207764
10. Bao L, Busany N, Lo D, Maoz S (2019) Statistical log differencing. In: ASE, pp 851–862
11. Barik T, DeLine R, Drucker S, Fisher D (2016) The bones of the system: A case study of logging and telemetry at microsoft. In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), IEEE, pp 92–101

12. Beschastnikh I, Liu P, Xing A, Wang P, Brun Y, Ernst MD (2020) Visualizing Distributed System Executions, vol 29. DOI 10.1145/3375633
13. Beschastnikh I, Liu P, Xing A, Wang P, Brun Y, Ernst MD (2020) Visualizing distributed system executions. TOSEM 29(2):1–38
14. Biermann A, Feldman J (1972) On the synthesis of finite-state machines from samples of their behavior. IEEE transactions on Computers 100(6):592–597
15. Bird C (2016) Interviews. In: Perspectives on Data Science for Software Engineering, Morgan Kaufmann
16. Broadfoot GH (2005) Asd case notes: Costs and benefits of applying formal methods to industrial control software. In: International Symposium on Formal Methods, Springer, pp 548–551
17. Buchbinder E (2011) Beyond checking: Experiences of the validation interview. Qualitative Social Work 10(1):106–122
18. Chen B, Jiang ZM (2021) A survey of software log instrumentation. ACM Computing Surveys (CSUR) 54(4):1–34
19. Chen B, et al (2017) Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation. Empirical Software Engineering 22(1):330–374
20. Cicchetti A, Di Ruscio D, Eramo R, Pierantonio A (2008) Automating co-evolution in model-driven engineering. In: 2008 12th International IEEE Enterprise Distributed Object Computing Conference, IEEE, pp 222–231
21. Dallmeier V, Knopp N, Mallon C, Hack S, Zeller A (2010) Generating test cases for specification mining. In: ISSTA, pp 85–96
22. Edwards D, Simmons S, Wilde N (2006) An approach to feature location in distributed systems. Journal of Systems and Software 79(1):57–68, DOI 10.1016/j.jss.2004.12.018
23. El-Masri D, Petrillo F, Guéhéneuc YG, Hamou-Lhadj A, Bouziane A (2020) A systematic literature review on automated log abstraction techniques. Information and Software Technology 122:106276
24. Etien A, Salinesi C (2005) Managing requirements in a co-evolution context. In: 13th IEEE International 31 Conference on Requirements Engineering (RE'05), IEEE, pp 125–134
25. Feng Y, Dreef K, Jones JA, van Deursen A (2018) Hierarchical abstraction of execution traces for program comprehension. In: ICPC, pp 86–96
26. Fidge CJ (1987) Timestamps in message-passing systems that preserve the partial ordering. Australian National University. Department of Computer Science
27. Fleming SD, Stirewalt R (2009) Successful strategies for debugging concurrent software: an empirical investigation. Michigan State University. Computer Science
28. Fluri B, Wursch M, PInzger M, Gall H (2007) Change distilling: Tree differencing for fine-grained source code change extraction. TSE 33(11):725–743
29. Flyvbjerg B (2007) Five Misunderstandings about Case-Study Research. Sage

30. Fu Q, Zhu J, Hu W, Lou JG, Ding R, Lin Q, Zhang D, Xie T (2014) Where do developers log? an empirical study on logging practices in industry. In: ICSE, pp 24–33

31. Gait J (1986) A probe effect in concurrent programs. Software: Practice and Experience 16(3):225–233

32. Gholamian S, Ward PA (2021) A comprehensive survey of logging in software: From logging statements automation to log mining and analysis. arXiv preprint arXiv:211012489

33. Gholamian S, Ward PA (2021) What distributed systems say: A study of seven spark application logs. In: 2021 40th International Symposium on Reliable Distributed Systems (SRDS), IEEE, pp 222–232

34. Goldstein M, Raz D, Segall I (2017) Experience report: Log-based behavioral differencing. In: ISSRE, pp 282–293

35. Graaf B, Lormans M, Toetenel H (2003) Embedded software engineering: the state of the practice. IEEE Softw 20(6):61–69

36. Gu S, Rong G, Zhang H, Shen H (2022) Logging practices in software engineering: A systematic mapping study. IEEE Transactions on Software Engineering

37. Gulzar MA, Zhu Y, Han X (2019) Perception and practices of differential testing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), IEEE, pp 71–80

38. Gupta M, Mandal A, Dasgupta G, Serebrenik A (2018) Runtime monitoring in continuous deployment by differencing execution behavior model. In: Pahl C, Vukovic M, Yin J, Yu Q (eds) ICSOC, Springer, Lecture Notes in Computer Science, vol 11236, pp 812–827, DOI 10.1007/978-3-030-03596-9_58, URL https://doi.org/10.1007/978-3-030-03596-9_58

39. Harty J, Zhang H, Wei L, Pascarella L, Aniche M, Shang W (2021) Logging practices with mobile analytics: An empirical study on firebase. In: 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), IEEE, pp 56–60

40. He S, He P, Chen Z, Yang T, Su Y, Lyu MR (2021) A survey on automated log analysis for reliability engineering. ACM Computing Surveys (CSUR) 54(6):1–37

41. Hendriks D, Meer Avd, Oortwijn W (2022) A multi-level methodology for behavioral comparison of software-intensive systems. In: International Conference on Formal Methods for Industrial Critical Systems, Springer, pp 226–243

42. Henzinger TA, Sifakis J (2007) The discipline of embedded systems design. Computer 40(10):32–40

43. Holton JA (2007) The coding process and its challenges. The Sage handbook of grounded theory 3:265–289

44. Hooimeijer B, Geilen M, Groote JF, Hendriks D, Schiffelers R (2022) Constructive Model Inference: Model learning for component-based software

architectures. In: 17th International Conference on Software Technologies (ICSOFT), pp 146–158

45. Jerding DF, Stasko JT, Ball T (1997) Visualizing interactions in program executions. In: ICSE, pp 360–370
46. Kabinna S, Bezemer CP, Shang W, Hassan AE (2016) Logging library migrations: A case study for the apache software foundation projects. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), IEEE, pp 154–164
47. Kabinna S, Bezemer CP, Shang W, Syer MD, Hassan AE (2018) Examining the stability of logging statements. Empirical Software Engineering 23(1):290–333
48. Kagdi H, Gethers M, Poshyvanyk D, Collard ML (2010) Blending conceptual and evolutionary couplings to support change impact analysis in source code. In: RE, pp 119–128
49. Krka I, Brun Y, Medvidovic N (2014) Automatic mining of specifications from invocation traces and method invariants. In: ESEC/FSE, pp 178–189
50. Kurfess TR, Hodgson TJ (2007) Metrology, sensors and control. In: Micromanufacturing, Springer, pp 89–109
51. Lee EA (2008) Cyber physical systems: Design challenges. In: 2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC), IEEE, pp 363–369
52. Legunsen O, Hassan WU, Xu X, Roşu G, Marinov D (2016) How good are the specs? a study of the bug-finding effectiveness of existing java api specifications. In: ASE, pp 602–613
53. Lemieux C, Park D, Beschastnikh I (2015) General ITL specification mining. In: ASE, pp 81–92
54. Levy O, Feitelson D (2019) Understanding large-scale software–a hierarchical view. In: ICPC, pp 283–293
55. Li H, Chen THP, Shang W, Hassan AE (2018) Studying software logging using topic models. EMSE 23(5):2655–2694
56. Li H, Shang W, Adams B, Sayagh M, Hassan AE (2020) A qualitative study of the benefits and costs of logging from developers' perspectives. TSE
57. Li S, Niu X, Jia Z, Liao X, Wang J, Li T (2020) Guiding log revisions by learning from software evolution history. Empirical Software Engineering 25(3):2302–2340
58. Lian X, Rahimi M, Cleland-Huang J, Zhang L, Ferrai R, Smith M (2016) Mining requirements knowledge from collections of domain documents. In: 2016 IEEE 24th International Requirements Engineering Conference (RE), pp 156–165
59. Liu X, Lin W, Pan A, Zhang Z (2007) WiDS checker: Combating bugs in distributed systems. 4th Symposium on Networked Systems Design and Implementation, NSDI 2007 pp 257–270
60. Lo D, Maoz S (2012) Scenario-based and value-based specification mining: better together. In: ASE, vol 19, pp 423–458

61. Lo D, Mariani L, Pezzè M (2009) 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, pp 345–354
62. Luo Q, Hariri F, Eloussi L, Marinov D (2014) An empirical analysis of flaky tests. In: FSE, pp 643–653
63. Maoz S, Ringert JO, Rumpe B (2010) A manifesto for semantic model differencing. In: MODELS, Springer, pp 194–203
64. Mashhadi MJ, Hemmati H (2019) An empirical study on practicality of specification mining algorithms on a real-world application. In: ICPC, pp 65–69
65. Mashhadi MJ, Siddiqui TR, Hemmati H, Loewen H (2019) Interactive semi-automated specification mining for debugging: An experience report. Information and Software Technology 113:20–38
66. Mattern F, et al (1988) Virtual time and global states of distributed systems. Univ., Department of Computer Science
67. McBurney PW, Liu C, McMillan C, Weninger T (2014) Improving topic model source code summarization. In: Proceedings of the 22nd international conference on program comprehension, pp 291–294
68. Mengerink J, Schiffelers RRH, Serebrenik A, van den Brand M (2016) Dsl/model co-evolution in industrial emf-based MDSE ecosystems. In: Mayerhofer T, Pierantonio A, Schätz B, Tamzalit D (eds) Proceedings of the 10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016), Saint-Malo, France, October 2, 2016, CEUR-WS.org, CEUR Workshop Proceedings, vol 1706, pp 2–7, URL http://ceur-ws.org/Vol-1706/paper1.pdf
69. Mishra D, Mishra A, Yazici A (2008) Successful requirement elicitation by combining requirement engineering techniques. In: 2008 First International Conference on the Applications of Digital Information and Web Technologies (ICADIWT), IEEE, pp 258–263
70. Noergaard T (2012) Embedded systems architecture: a comprehensive guide for engineers and programmers. Newnes
71. Pandey D, Suman U, Ramani AK (2010) An effective requirement engineering process model for software development and requirements management. In: 2010 International Conference on Advances in Recent Technologies in Communication and Computing, IEEE, pp 287–291
72. Pecchia A, Cinque M, Carrozza G, Cotroneo D (2015) Industry practices and event logging: Assessment of a critical software development process. In: ICSE (2), IEEE, pp 169–178
73. Pradel M, Gross TR (2012) Leveraging test generation and specification mining for automated bug detection without false positives. In: ICSE, pp 288–298
74. Rong G, Zhang Q, Liu X, Gu S (2017) A systematic review of logging practice in software engineering. In: 2017 24th Asia-Pacific Software Engineering Conference (APSEC), IEEE, pp 534–539

75. Rong G, Gu S, Zhang H, Shao D, Liu W (2018) How is logging practice implemented in open source software projects? a preliminary exploration. In: 2018 25th Australasian Software Engineering Conference (ASWEC), IEEE, pp 171–180
76. Rong G, Xu Y, Gu S, Zhang H, Shao D (2020) Can you capture information as you intend to? a case study on logging practice in industry. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 12–22
77. Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering 14(2):131
78. Said W, Quante J, Koschke R (2018) Towards interactive mining of understandable state machine models from embedded software. In: MODELSWARD, pp 117–128
79. Seaman CB (1999) Qualitative methods in empirical studies of software engineering. IEEE Transactions on software engineering 25(4):557–572
80. Shaft TM, Vessey I (1995) The relevance of application domain knowledge: the case of computer program comprehension. ISR 6(3):286–299
81. Shang W, Nagappan M, Hassan AE, Jiang ZM (2014) Understanding log lines using development knowledge. In: ICSME, pp 21–30
82. Shull FJ, Carver JC, Vegas S, Juristo N (2008) The role of replications in empirical software engineering. Empirical software engineering 13(2):211–218
83. da Silva AJ, Linhares MV, Padilha R, Roqueiro N, de Oliveira RS (2006) An empirical study of sysml in the modeling of embedded systems. In: 2006 IEEE International Conference on Systems, Man and Cybernetics, IEEE, vol 6, pp 4569–4574
84. Silva E, Freitas EP, Wagner FR, Carvalho FC, Pereira CE (2006) Java framework for distributed real-time embedded systems. In: Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06), IEEE, pp 8–pp
85. Storey MA, Ernst NA, Williams C, Kalliamvakou E (2020) The who, what, how of software engineering research: a socio-technical framework. Empirical Software Engineering 25(5):4097–4129
86. Strandberg PE, Enoiu EP, Afzal W, Sundmark D, Feldt R (2019) Information flow in software testing–an interview study with embedded software engineering practitioners. IEEE Access 7:46434–46453
87. Strauss A, Corbin JM (1997) Grounded theory in practice. Sage
88. Tan L, Bockisch C (2019) A survey of refactoring detection tools. In: EMLS, pp 100–105
89. Vogel-Heuser B, Fay A, Schaefer I, Tichy M (2015) Evolution of software in automated production systems: Challenges and research directions. Journal of Systems and Software 110:54–84
90. Walkinshaw N, Taylor R, Derrick J (2016) Inferring extended finite state machine models from software executions. Empirical Software Engineering 21(3):811–853

91. van der Werf JMEM, van Dongen BF, Hurkens CAJ, Serebrenik A (2009) Process Discovery using Integer Linear Programming. Fundam Inform 94(3-4):387–412

92. Yang N, Aslam K, Schiffelers R, Lensink L, Hendriks D, Cleophas L, Serebrenik A (2019) Improving model inference in industry by combining active and passive learning. In: SANER, pp 253–263

93. Yang N, Cuijpers PJL, Schiffelers RRH, Lukkien J, Serebrenik A (2021) An interview study of how developers use execution logs in embedded software engineering. In: 43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021, IEEE, pp 61–70, DOI 10.1109/ICSE-SEIP52600.2021.00015, URL `https://doi.org/10.1109/ICSE-SEIP52600.2021.00015`

94. Ye X, Bunescu R, Liu C (2014) 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, pp 689–699

95. Yuan D, Park S, Zhou Y (2012) Characterizing logging practices in open-source software. In: 2012 34th International Conference on Software Engineering (ICSE), IEEE, pp 102–112

96. Zaidman A, Demeyer S (2004) Managing trace data volume through a heuristical clustering process based on event execution frequency. In: CSMR, pp 329–338

97. Zaidman A, Van Rompaey B, van Deursen A, Demeyer S (2011) Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Software Engineering 16(3):325–364

98. Zeng Y, Chen J, Shang W, Chen THP (2019) Studying the characteristics of logging practices in mobile apps: a case study on f-droid. Empirical Software Engineering 24(6):3394–3434

99. Zhu J, He P, Fu Q, Zhang H, Lyu MR, Zhang D (2015) Learning to log: Helping developers make informed logging decisions. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE, vol 1, pp 415–425