An interview study of how developers use execution logs in embedded software engineering

Nan Yang
Pieter Cuijpers
Eindhoven University of Technology
The Netherlands

Ramon Schiffelers
ASML, Eindhoven University of Technology
The Netherlands

Johan Lukkien
Alexander Serebrenik
Eindhoven University of Technology
The Netherlands

***Abstract*—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 analyze logs in embedded software engineering.**

In order to bridge the gap, we study how developers analyze logs by interviewing 25 software developers from ASML, which is a leading company in developing lithography machines. In particular, we explore 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. As the main contribution, 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, we observed that inspecting execution information at different levels of abstraction is useful to develop comprehension of such complex systems. However, obtaining the abstraction is difficult with current tools. Our study has several implications. The empirical evidence provided by our study implies the need to support log inspection and comparison with multiple levels of abstraction, categorize log differences, and recover links between different types of logs.

***Index Terms*—log analysis, embedded systems, maintenance**

I. 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 [38], test generation [36], and change comprehension [5]. However, researchers have provided empirical evidence that some log analysis tools are not necessarily effective and applicable when dealing with real-world problems [33].

We believe that understanding how developers analyze logs is essential to design better log analysis tools. Li et al. [29] 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. [6] identified the tensions that emerge in data-driven cultures as event data 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 data. Different from Li et al. and Barik et al., we focus on how developers analyze logs in embedded software engineering, with the aim of identifying developers' needs for future research on the tools that are applicable to aid developers in performing their maintenance tasks.

We would like to thank ASML for supporting this research.

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 developers expect from tools (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 of all, 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, we conducted an exploratory case study at ASML which 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, throughput information as well as business-critical data (RQ1). We confirmed that logs are primarily used for analyzing software issues [6], [29]. 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 [29], [47], 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). Based on our findings, we proposed a set of suggestions.

II. Methodology

To understand how software developers use logs in the embedded systems industry, we conducted a case study [37]. As our research questions differ from previous work [6], [29], we opted for an exploratory rather than a confirmatory study.

A. 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 [25]. The software components developed by this division form a paradigmatic subsystem [15] that coordinate machine actions and measurements as well as calibration of the systems based on the performed measurements. The subsystem consists of multiple processes collaborating with each other via inter-process communication. Moreover, 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 [3], 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.

B. 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 [8]. Table I shows our interview guide. In adherence to the best interviewing practices [8], we conducted a pilot interview with a developer from the same division. 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 our university and the participating company.

1) 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 II). 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) Data collection and analysis: We collected data by recording the audio and making the transcripts. We coded the transcripts [8] using the *ATLAS.ti* data analysis software. Our coding process consists of three steps. First, we performed open coding. We constantly compared and refined codes that emerge from this process. Similar codes were then grouped into categories. Second, we conducted axial coding to make connections between codes or categories. Finally, these codes and categories were grouped into the topics derived from our research questions. According to Strauss and Corbin [41], 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.

TABLE I: 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?

3) Member checking: Coding is an interpretative process and as such there is always a risk of misinterpretation [21]. In order to reduce this risk, we performed member checking [9], 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 has been 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.

III. Type of logs (RQ1)

The types of execution logs are summarized in Table III.

TABLE II: Demographics of interviewees

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

TABLE III: Four types of logs (RQ1)

Type	Information	Enabled by default	Presence of physical layers
Event log (EL)	Machine event & error message	✓	Not necessary
Function trace (FT)	Order of functions & values of parameters	✗	Not necessary
Throughput trace (TT)	Duration of software & hardware actions	✓	Necessary
Functional data (FD)	Business-critical data	✓	Not necessary

Event logs contain regular events created when a machine action such as initialization has been executed as well as error messages of the systems. 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 trace shows 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). 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). 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). Interviewees use "tracing", "function trace" or "component tracing" interchangeably.

Since the production throughput is a key business driver of the machines, **throughput trace** 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). The throughput trace logs the duration and sequence of software and hardware actions, showing the *speed* of execution. Obtaining throughput traces 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).

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). 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.

TABLE IV: The purposes of log analysis and the used logs for those purposes. * indicates the purposes that were not identified in the previous work. "#I" indicates the number of interviewees who mention the purpose during interviews.

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

IV. Purpose of log analysis (RQ2)

A. Purposes

Table IV shows the purposes for which developers analyze execution logs. We identified 18 purposes and classified them

into four categories. Our findings are complementary to the prior studies [6], [29], [46]. 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.

1) ***Software comprehension****:* 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). 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).

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) ***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). 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). In order to achieve high throughput performance, execution logs are also used to verify if actions are finished within time budget 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). 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).

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. By analyzing logs, developers also get a rough idea of whether the issue analyzed is caused by their components or not, and which group or person has the expertise to fix the issue. Furthermore, log analysis helps developers identify the root cause of the issue. Particularly, analyzing the logs provided by customers is essential for identifying the root cause of the field issues, a prerequisite for reproducing the issues on the in-house machines. Based on logs, developers can further confirm if issues are correctly reproduced. 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 log, 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). For issues found by testing, developers analyze logs to identify why a software change does not pass tests and why the execution outcomes of tests against the same code appear not deterministic [32].

B. 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 provides the low-level execution details of components. Function data are particularly used for issue analysis and throughput traces for performance-related purposes. A closer look at Table IV 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). Moreover, debuggers can interfere with timing behaviour and synchronisation between multiple processes: "*What might happen is that you have some timeout, so some processes hanging waiting for the process you are debugging. If he doesn't answer in a short time, it stops. Basically it throws an error*" (P24). This requires developers 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). 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 completely different domain. For example we need that much better [functional] data logging*". 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).

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).

V. Information needs (RQ3)

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

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).

First of all, developers inspect event logs and functional data to know the settings (e.g., user inputs) of the software. 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 are based on the Client-Server architecture [34]. 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.

To further understand 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).

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 machine to us sometimes is 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). 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 function, is as you expected*" (P22).

3) ***Software state and interaction****:* To understand the software system, developers analyze the interactions between software components based on the function traces. 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. The division adopts model-driven engineering (MDE) to design the components that are responsible for controlling machine actions and production processes. Developers design these components using state machines and generate code automatically from these state machines. Each control component 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 software, 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).

4) ***Timing performance****:* Developers analyze throughput traces to understand and 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).

5) ***Differences between executions****:* Developers need the information about the differences between the logs generated from multiple executions in order to, e.g., identify regression, and understand software change. This is especially the case for flaky tests [32]: "*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).

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 complex systems. Particularly, developers need the *differences between executions* for identifying the cause of flaky tests or the deviation from expected behavior.

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

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

VI. 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.

A. 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). 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). 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) or use text difference analyzers (e.g., KDiff3, beyond compare and Linux diff): "*Sometimes I use beyond comparison for comparing logs. It compares data line by line*" (P2).

B. Challenges in log analysis

Table VI summarizes the challenges identified.

1) ***Log availability and quality****:* In order to enable log analysis, developers first need to collect logs. As mentioned in Section III, due to the needs of 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); 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). The quality of logs is also known to influence the developers' ability to perform the analysis efficiently [16], [28], [47]. 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). 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) or scant (P9).

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

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). 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).

2) ***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*". For the components that are responsible for process control and implemented using interacting state machines (cf. Section V-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).

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).

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 sub-processes...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). 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 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). Similarly, irrelevant differences can be introduced by concurrency: "*Some events are not necessarily happening in the same order in different executions*" (P11). 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), effort and time: "*more and more preprocessing until you remove the most of them...It costs time. And it can even lead you to wrong conclusions*" (P17).

3) ***Expertise****:* The systems are not only complex but also multidisciplinary. Working with logs generated from such systems requires domain knowledge (e.g., how machines expose wafers to the light): "*We can dive into the trace files etc. It is not enough. You have to know what is actually going on here with those traces and what is the component doing*" (P11). 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). 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 but also 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).

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). 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 callback]*" (P15).

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

Challenges	#I
Log availability and quality	
Absence of logs	8
Non-standard logging	5
Incompleteness of trace	8
Presence of noise	18
Unreadable tracing format for functions with a lot of parameters	2
Missing categorization and overview	3
Broken error linking	4
Complexity	
Involvement of components from different groups and domains	6
Involvement of many state machines	2
Presence of concurrency	8
Presence of irrelevant differences between logs	5
Expertise	
Lack of domain knowledge	10
Unfamiliar with code base and software design	9

C. Expected tools

1) Creating multi-level abstraction: Developers would like to have a tool that can help them inspect different levels of details from logs: "*On certain levels you can open and close those functions to see what's internally there so that you can maintain a high level overview and details where you need them, instead of only having all the details now, but that's what's happening now, you got whole bunch of data and it's all detail*" (P14). 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). 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). For example, as discussed in Section V-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 visualising the sequence diagrams only for these important interactions.

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). Furthermore, developers envision tools supporting identification of the cause(s) of log differences such as concurrency, refactoring or uninitialised variables.

2) Providing generic and unified facilities: Instead of multiple scripts with (partially) duplicated functionality, develop-

ers 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), 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). 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).

RQ4 summary: Developers mainly use textual-based tools to analyze logs. In addition to log quality concerns, *concurrency* and *irrelevant differences between logs* bring additional challenges in log analysis. Developers 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.

VII. Discussion and implications

Our study of log analysis in embedded software engineering *not only* provides empirical evidence consistent with general software engineering studies *but also* highlights the insights obtained exclusively from our problem context. We start by relating our findings to literature (Section VII-A), and then discuss implications of our work (Section VII-B).

A. Discussion

1) ***Software concurrency****:* Logs are heavily used by developers because of the difficulties of using a traditional debugging approach for concurrent programs. 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 [17]. We also observed that interleaving of concurrent executions incurs challenges not only in program comprehension [2] but also in log comprehension (cf. Section VI-B2).

2) ***Expertise and information needs****:* As discussed in Section VI-B3, lack of familiarity with the existing code can hinder understanding logs. This concurs with a recent study of confusion in code reviews [12]. We learned that domain knowledge is essential, 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. This observation is consistent with the earlier findings [19].

We observed that experienced developers tend to adopt a top-down approach when inspecting logs. This concurs with the study on the relevance of application domain knowledge in program comprehension—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 [39]. The top-down method is known to be effective for system comprehension which requires developers understand the structure of the system [27]. 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 (cf. Section V).

Our work is also related to the study of information needs by Shang et al. [40], that have studied log-related information needs of the *users* (i.e., operators and administrators). We have taken a complementary perspective and focused on information needs of *developers*. As opposed to users, developers, responsible for maintaining the code base, not only need the diagnostic information (e.g., the context of error messages) but also execution details (e.g., interactions between components).

3) ***Logging quality and practice****:* Among the challenges identified in Table VI, seven are related to log availability and quality. Availability and quality of logs have attracted significant attention from the research community that studied such questions as where to log [16], [28], what to log [47], how to log [10] and how to use logs [20]; and such challenges as absence of logs [29], non-standard logging [35], and presence of noise and incomplete logging [29].

Extending this body of work we discovered several issues that hinder log analysis and have not yet been discussed in the scientific literature. If the log does not contain desired information, merely recollecting the log is non-trivial for manufacturing systems as this would require hardware components or simulation. Furthermore, the metrology software, where functions are usually augmented with complex data structures, needs better logging format of function traces. We also found that the company uses an error linking mechanism to log the propagation of errors across components. The presence of implementation bugs that break the error link shown in logs increases the difficulties of issue analysis with logs, which requires a customized detector to prevent such bugs.

B. Implications

Above we discussed the information needs (Section V), challenges (Section VI-B) and expectations (Section VI-C) of log analysis tools. Consistent with the previous study at Microsoft [6], we found that developers use text editors for their log analysis activities. Given that many log analysis tools have been proposed over years, the observation implies a gap between research prototypes and industrial practice. Below we extend the discussion with the implications of how the existing research work can help developers and what further improvements tool builders and researchers can consider.

*1) **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 [24], [30], [43], sets of temporal properties [26], and execution patterns [45]. This kind of trace abstraction tools often rely on heuristics to create abstraction, which can result in overgeneralization or undergeneralization [44]. Moreover, these tools provide only one level of abstraction, not meeting the expectations of the interviewees (cf. Section VI-C1). Several studies addressed this limitation [7], [13], [22] by allowing developers inspect information at different levels of details. 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 [7], [22]. Tool builders may take the context of use into account, incorporating information from other sources (e.g., bug reports) to guide developers navigate through information at different abstractions for their tasks.

Several tools can compare behavioral models extracted from logs generated from multiple executions [1], [5], [18]. However, these tools 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. Researchers may consider to improve the underlying techniques with intuitive user configurations.

*2) **Categorizing log differences to support evolution tasks**:* To support developers in understanding differences between the logs, researchers and tool builders can consider identifying and categorizing the differences according to such causes as concurrency, refactoring or functional changes. Categorizing the differences can help developers perform evolution tasks: e.g., when identifying a root cause of regression based on logs, developers can ignore the differences belonging to the categories of concurrency and refactoring because these differences are not expected to influence the final outcome.

To identify log differences related to concurrency tool builders can leverage previous work on log analysis for automatic detection of concurrency bugs [11], [31]. To recognize the differences caused by code modifications such as refactoring and functional modifications, tool builders may consider to leverage the existing tools from the field of code differencing [14] and refactoring detection [42]. The obtained information can be incorporated into log comparison to help developers recognize the useful log differences for their tasks.

*3) **Linking different types of logs to obtain a complementary picture of executions**:* Such complex systems generate different types of logs that capture different aspects of executions (cf. Section III), requiring developers to recover the links between them (cf. Section V-1 and VI-C2). To help developers obtain a complementary picture of an execution, the tool builders can consider recovering the links between different types of logs, e.g., using the timing information. 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 [4] and change impact analysis [23].

VIII. Threats to validity

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

Threats to **internal validity** concern factors that might have influenced the results. First, developers might misunderstand our interview questions. We mitigated this risk by piloting the interview with a developer from the division, and rewording the questions as necessary. Second, our interviewees might hesitate to discuss the difficulties in their current practice or the issues in the tools they use, as they were aware that the result will be published. We reduced their concern by explaining data privacy rights and guaranteeing the 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. 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.

Threats to **external validity** concern the generalizability of our conclusions beyond the studied context. We opted for convenience sampling selecting the company that we have the 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 [19]. 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 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. However, we acknowledge that the external validity of this interview study at a single company is inevitably limited. We consider our interviews as an exploratory study for which we reported observations rather than facts, e.g., we do not claim that *all* embedded developers use text-based tools for log analysis based on our observations from a single company. To obtain conclusive evidence and build theories, confirmatory studies at multiple companies are required.

IX. Conclusion

We explored how developers use logs in embedded software engineering by interviewing 25 developers from ASML. We identified four types of logs developers use, 18 purposes for which developers use logs and 13 types of information developers search in logs. The most prevalent types of information are *propagation of errors across systems*, *timestamp associated with log lines*, *data flow*, *interaction of software modules*, and *differences between multiple executions*. 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. Particularly, we found that inspecting execution information at different levels of abstraction is useful to develop global comprehension. However, obtaining the abstraction is difficult with currently used tools. Our study has several implications. Log analysis tools could be improved by supporting multi-level abstraction for log inspection and comparison, categorizing log differences and recovering links between different types of logs.

References

[1] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. Using finite-state models for log differencing. In *ESEC/FSE*, pages 49–59, 2018.

[2] Cyrille Artho, Klaus Havelund, and Shinichi Honiden. Visualization of concurrent program executions. In *COMPSAC*, pages 541–546, 2007.

[3] Sara Abbaspour Asadollah, Rafia Inam, and Hans Hansson. A survey on testing for cyber physical system. In *ICTSS*, pages 194–207, 2015.

[4] Hazeline Asuncion, Arthur Asuncion, and Richard Taylor. Software traceability with topic modeling. In *ICSE (1)*, pages 95–104, 2010.

[5] Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. Statistical log differencing. In *ASE*, pages 851–862, 2019.

[6] Titus Barik, Robert DeLine, Steven Drucker, and Danyel Fisher. The bones of the system: A case study of logging and telemetry at microsoft. In *ICSE-C*, pages 92–101, 2016.

[7] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D Ernst. Visualizing distributed system executions. *TOSEM*, 29(2):1–38, 2020.

[8] Christian Bird. Interviews. In *Perspectives on Data Science for Software Engineering*. Morgan Kaufmann, 2016.

[9] Eli Buchbinder. Beyond checking: Experiences of the validation interview. *Qualitative Social Work*, 10(1):106–122, 2011.

[10] Boyuan Chen and Zhen Ming Jiang. Characterizing and detecting anti-patterns in the logging code. In *ICSE*, pages 71–81, 2017.

[11] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *CCS*, pages 1285–1298, 2017.

[12] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion in code reviews: Reasons, impacts, and coping strategies. In *SANER*, pages 49–60, 2019.

[13] Yang Feng, Kaj Dreef, James A Jones, and Arie van Deursen. Hierarchical abstraction of execution traces for program comprehension. In *ICPC*, pages 86–96, 2018.

[14] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *TSE*, 33(11):725–743, 2007.

[15] Bent Flyvbjerg. *Five Misunderstandings about Case-Study Research*. Sage, 2007.

[16] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *ICSE*, pages 24–33, 2014.

[17] Jason Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3):225–233, 1986.

[18] Maayan Goldstein, Danny Raz, and Itai Segall. Experience report: Log-based behavioral differencing. In *ISSRE*, pages 282–293, 2017.

[19] Bas Graaf, Marco Lormans, and Hans Toetenel. Embedded software engineering: the state of the practice. *IEEE Softw.*, 20(6):61–69, 2003.

[20] Monika Gupta, Atri Mandal, Gargi Dasgupta, and Alexander Serebrenik. Runtime monitoring in continuous deployment by differencing execution behavior model. In *ICSOC*, pages 812–827, 2018.

[21] Judith A Holton. The coding process and its challenges. *The Sage handbook of grounded theory*, 3:265–289, 2007.

[22] Dean F Jerding, John T Stasko, and Thomas Ball. Visualizing interactions in program executions. In *ICSE*, pages 360–370, 1997.

[23] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *RE*, pages 119–128, 2010.

[24] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *ESEC/FSE*, pages 178–189, 2014.

[25] Thomas R Kurfess and Thom J Hodgson. Metrology, sensors and control. In *Micromanufacturing*, pages 89–109. Springer, 2007.

[26] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. General ITL specification mining. In *ASE*, pages 81–92, 2015.

[27] Omer Levy and Dror Feitelson. Understanding large-scale software–a hierarchical view. In *ICPC*, pages 283–293, 2019.

[28] Heng Li, Tse-Hsun Peter Chen, Weiyi Shang, and Ahmed E Hassan. Studying software logging using topic models. *EMSE*, 23(5):2655–2694, 2018.

[29] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. A qualitative study of the benefits and costs of logging from developers' perspectives. *TSE*, 2020.

[30] David Lo and Shahar Maoz. Scenario-based and value-based specification mining: better together. In *ASE*, volume 19, pages 423–458, 2012.

[31] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *ESEC/FSE*, pages 3–14, 2018.

[32] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, 2014.

[33] Mohammad Jafar Mashhadi and Hadi Hemmati. An empirical study on practicality of specification mining algorithms on a real-world application. In *ICPC*, pages 65–69, 2019.

[34] Tammy Noergaard. *Embedded systems architecture: a comprehensive guide for engineers and programmers*. Newnes, 2012.

[35] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. Industry practices and event logging: Assessment of a critical software development process. In *ICSE (2)*, pages 169–178. IEEE, 2015.

[36] Michael Pradel and Thomas R Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE*, pages 288–298, 2012.

[37] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *EMSE*, 14(2):131, 2009.

[38] Wasim Said, Jochen Quante, and Rainer Koschke. Towards interactive mining of understandable state machine models from embedded software. In *MODELSWARD*, pages 117–128, 2018.

[39] Teresa M Shaft and Iris Vessey. The relevance of application domain knowledge: the case of computer program comprehension. *ISR*, 6(3):286–299, 1995.

[40] Weiyi Shang, Meiyappan Nagappan, Ahmed E Hassan, and Zhen Ming Jiang. Understanding log lines using development knowledge. In *ICSME*, pages 21–30, 2014.

[41] Anselm Strauss and Juliet M Corbin. *Grounded theory in practice*. Sage, 1997.

[42] Liang Tan and Christoph Bockisch. A survey of refactoring detection tools. In *EMLS*, pages 100–105, 2019.

[43] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. *EMSE*, 21(3):811–853, 2016.

[44] Nan Yang, Kousar Aslam, Ramon Schiffelers, Leonard Lensink, Dennis Hendriks, Loek Cleophas, and Alexander Serebrenik. Improving model inference in industry by combining active and passive learning. In *SANER*, pages 253–263, 2019.

[45] Andy Zaidman and Serge Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR*, pages 329–338, 2004.

[46] Yi Zeng, Jinfu Chen, Weiyi Shang, and Tse-Hsun Peter Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *EMSE*, 24(6):3394–3434, 2019.

[47] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *ICSE*, pages 415–425, 2015.