Encoding domain knowledge of logs during log analysis

Zamfirov, Filip

Award date:
2023

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Software Engineering and Technology

Encoding domain knowledge of logs during log analysis

Master Thesis

Filip Zamfirov

Supervisors:
prof. dr. Alexander Serebrenik
dr. Dennis Dams

Committee:
prof. dr. Alexander Serebrenik
dr. Dennis Dams
dr. Stef van den Elzen
dr. Ivan Kurtev

Version 1.0

Eindhoven, August 2023

Abstract

With software permeating our world, modern software systems grow both in size (e.g., lines of code, number of software artefacts) and complexity. Logs produced by software systems during execution are regularly the first and only information used by software engineers to comprehend the behaviour of these complex systems and to repair software faults. However, analysing logs is not trivial due to the immense size of the information logged by these systems. Many previous studies have proposed tools and techniques aiming to support log analysis, despite that, software developers commonly rely on text-based tools as well as self-made scripts and programs when analysing logs. The functionalities of such tools are often limited and do not provide developers with a way to incorporate their domain knowledge into the analysis. The domain knowledge of developers lives mainly in their minds.

The goal of this thesis is to investigate how to enable software engineers to better utilise their domain knowledge of logs more actively in the analysis process. Specifically, we investigate the presence of patterns and sequences which we refer to as structures as well as the knowledge of these structures. Furthermore, as logs are commonly analysed in text-based tools in which they are represented as raw textual data, we aim to make log analysis more visual and interactive.

To that end, we conduct an interview study in Philips Image Guided Therapy Systems (IGTS) - a leading company specialising in the field of health technology, to understand what structures are present in logs and whether they are utilised by software engineers during log analysis. As a result, we observe that software engineers often utilise their domain knowledge of specific structures occurring in the logs to distinguish between irrelevant and relevant, for their analysis, parts. Consequently, we develop functionality that enables developers to encode their domain knowledge of structures in log analysis by defining structures and searching for them in logs visually and interactively. We evaluate our implementation with software engineers on logs produced by industrial software. We discover that the participants are able to encode their knowledge of structures successfully and use our features to facilitate the investigation of software issues. However, we also discovered some limitations of the features. First, the usefulness of the features depends on the experience and knowledge of the users. The features may not be helpful for analysing unfamiliar logs. Second, the features may not handle well structures that contain a large number of entries. Specifically, the results may be hard to navigate and use. Despite these limitations, our solution receives above-average ratings for usability from the evaluation participants.

Preface

First and foremost, I would like to express my deepest gratitude to my supervisors, dr. Alexander Serebrenik and dr. Dennis Dams for their guidance and support during this project. Not only did they provide me with invaluable advice and encouragement, but they have been genuine role models whom I admire, and whose high standards I have striven to meet. I consider myself fortunate to have had the opportunity to learn from both of them.

I am very grateful to Christos Kitsanelis for his continued support and words of encouragement. Additionally, within Philips IGTS, I would like to thank Lena Filatova for all of her assistance as well as everyone who took the time and shared their knowledge on the topic of log analysis or provided feedback on the implementation of this work.

Furthermore, I am grateful to my assessment committee members dr. Stef van den Elzen and dr. Ivan Kurtev for reviewing this thesis and attending the defence.

Finally, I am indebted to my partner, my parents and my brother for their unreserved love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, software systems are employed in a variety of fields and forms, ranging from simple mobile apps used daily to complex cybersecurity solutions used by large organisations. Due to this ever-expanding presence of software, software systems evolve both in functionality and in complexity [1, 2, 3]. However, as their complexity increases, so does the difficulty of comprehending their behaviour. One way in which software developers can better grasp the behaviour of complex systems is by analysing software logs - a collection of recorded events that occur in a software system during execution. These logs can vary in the shape and size of their contents, in format and in purpose, among other factors. These files are often the main information source on software behaviour [4], and are analysed for various purposes, with the most commonly cited one being the investigation of software bugs[1][5].

In the context of this study, we consider two common categories of logs - event logs and functional traces [5]. Event logs often record general information at a high level, such as an entire software (sub)system, whereas traces are more detailed and record information at the level of sub-units (e.g., components, classes). As one software operation or action can be completed in milliseconds, new information can be logged every few milliseconds. Therefore, even if a system runs for several minutes, the recorded, in both event logs and functional traces, information can consist of thousands of entries or more. For instance, Google systems generate hundreds of millions of entries monthly, consuming tens of terabytes daily [6, 7]. Thus, obtaining specific information manually can be significantly time-consuming and tedious. To improve this process, various tools and methodologies have been proposed [8, 9, 10, 11, 12, 13]. Despite the differences between event logs and traces, proposed methodologies and tools used for their analysis are very similar. Regardless, such tools and techniques are seldom used in the software industry as developers mainly rely on unspecialised tools such as text editors and self-made scripts [5]. One possible explanation for this limited adoption might be the limitations and prerequisites of the state-of-the-art tools. Some of these tools require modifying the source code to produce logs to be used as input [14]. Such modifications can require excessive amounts of work due to the size and complexity of software. Other tools accept event logs and traces in specific formats [11], which can be too restrictive (e.g., contents-wise and size-wise). Finally, an extensive setup or configuration might be needed before a tool can be used [8]. Such factors can discourage the use of state-of-the-art tools and methodologies. As a result, many industry professionals still analyse event logs and traces manually or with minimal tool support, often in the form of keyword searching provided by text editors, and/or filtering on field values (e.g., Timestamp, entry description. message, etc.). Furthermore, the data in logs and traces is commonly repres-

[1] also referred to in the literature as software faults

ented as millions of lines of text. Processing such voluminous data is both very time-consuming and tedious. The development of more intuitive and easy-to-use tools may be one approach to alleviate this problem. Furthermore, event log and functional trace analysis could be accelerated by representing the logged information in different views and enabling the users to interact with them [12, 9]. Such views may consist of textual representations which aggregate and abstract information or advanced visualisations that condense thousands of entries into intuitive and interactable (e.g., filter, resize, search, etc.) figures and plots. Other techniques, such as (automated) stream abstractions [4], can facilitate the comprehension of the information in the logs by reducing the amount of data to process. This thesis investigates whether the domain knowledge about the contents of event logs and traces, that developers possess, can be used to provide a visual and interactive event log and functional trace analysis. For the sake of brevity, in the following sections, we shall refer to both event logs and functional traces as logs, unless a distinction is required.

1.1 Thesis context

Previous studies have explored ways to support log analysis by incorporating concepts such as patterns [14], intervals [11] and motifs [15] into the analysis. While these terms are closely related, there can be semantic differences between them. In this study, we opt to use the term *structure* to encapsulate the aforementioned concepts. The large recorded data in logs often contains inherent structures. Some structures can be explicit, having a clear beginning and end, visible in the log. An example of such a structure could be the information logged during a function execution. In this case, the structure would consist of two entries marking the beginning and the return of the function, as well as all logged entries in between. Figure 1.1 presents an example of such a structure. A structure could be also implicit. For example, a system that is being turned on must go through the same startup steps. In this, case the entries logged during this startup can be considered as an implicit structure (Figure 1.2). Domain knowledge of logs, such as the presence of structures, lives in the minds of developers experienced with the source code producing the logs [16].

The goal of this thesis is to observe what types of structures exist in logs, in particular structures of which developers are aware. Then given these structures, to investigate whether the domain knowledge about them can be leveraged during log analysis in order to facilitate it.

```
# Timestamp      Type    Thread ID       Location        Message
...
2023-03-15 09:43:04.576 >>>     8884    src\main.cpp:31 enter::shutDown
2023-03-15 09:43:04.576 >>>     8884    src\connectionManager.cpp:15    enter::closeConnection, connectionId= 1
2023-03-15 09:43:04.576 <<<     8884    src\connectionManager.cpp:21    exit::closeConnection, connectionId= 1, connectionClosed= True
2023-03-15 09:43:04.576 Info    8884    src\main.cpp:550        Destroy process handler
2023-03-15 09:43:04.576 Info    8884    src\main.cpp:553        Process handler argument # 0 : " PH02 "
2023-03-15 09:43:04.576 Info    8884    src\system.cpp:72       FUAP destroyed 13 2022 @ 07:06:29
2023-03-15 09:43:04.576 <<<     8884    src\main        exit::shutDown
...
```

shutDown function

Figure 1.1: Example of an explicit structure in a log

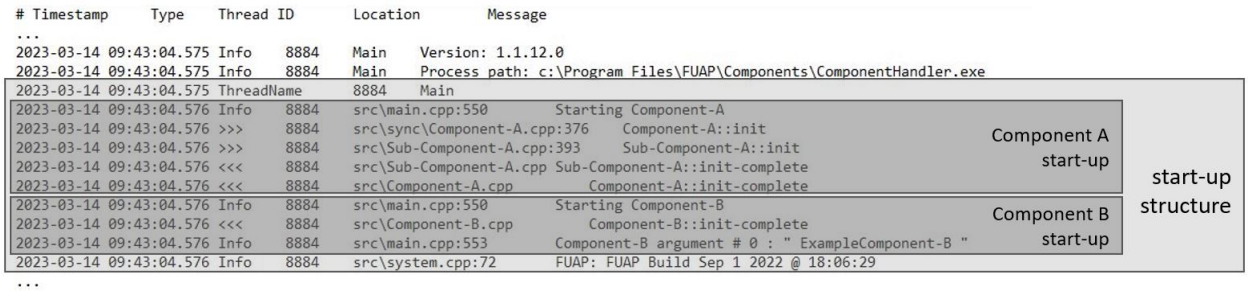

Figure 1.2: Example of an implicit structure in a log

1.2 Research questions

Before considering how the process of log analysis can be improved, first, we need to understand what structures are present in analysed files and what is their current impact on log analysis (RQ1) While there can exist many collections of log entries which can be considered as a structure, we are interested in structures which are known by developers, as those could possibly be utilised to facilitate the analysis process. Obtaining relevant information easily with tools would allow developers to direct their effort on the engineering work, rather than on looking through the logs and traces. Therefore, we need to understand what tool support can exploit the knowledge of these structures (RQ2) to achieve this goal. Hence, we pose the following research questions:

RQ1: What structures are present in event logs and traces analysed at Philips?

RQ2: Given these structures, what tool features can support an interactive analysis of these event logs and traces?

As the objective of this study is to ultimately improve the experience of performing log and trace analysis, we are interested in the impact of features described as an answer to the previous research question. Thus, we lastly pose this question:

RQ3: How do these tool features impact log analysis?

1.3 Case study: Philips

To address our research questions we conduct an exploratory case study [17] in Philips Image Guided Therapy Systems (IGTS). While previous studies have investigated the exploitation of patterns, intervals and similar concepts in log analysis, many such studies focus on techniques for automatic pattern detection [15] while others lack visualisations and interactivity [11]. Other previous works that discuss visualising and interacting with patterns consider analysis on collections of logs [9, 12, 13] and are not meant for the analysis of a single log such as the one commonly performed by software engineers. In Chapter 2, we present the aforementioned studies. To the best of our knowledge, there are no previous studies that investigate how knowledge of such concepts can be encoded in a visual and interactive way, to facilitate the analysis of logs for the purpose of software fault analysis, in an industrial setting.

Philips IGTS provides software and services for interventional imaging systems and smart devices used in the field of healthcare. As such, robustness and reliability are crucial for their systems and products. Therefore, Philips IGTS has defined internal procedures for finding software bugs and resolving them. To that end, event logs and traces are regularly used in the context of investigating software bugs, both internally reported (referred to as engineering issues or EIs) and field issues (referred to as product defects). However, processing relevant logs and localising the specific information leading to the source of the bug can be very time-consuming. While some software engineers within Philips IGTS have developed self-made tools to visualise and facilitate the processing of logs, frequently developers resort to re-implementing similar features, with only minor variations. Hence, a generic solution applying advanced techniques is needed. We believe that as a leading technological company in the field of healthcare, Philips IGTS provides a suitable context for conducting our investigation.

1.4 Thesis outline

The remainder of this study is structured as follows. Chapter 2 presents some relevant background on log entries and discusses relevant literature by separating the related tools and techniques on methodology. Then in Chapter 3, we discuss the methodology and results of our interview study. Chapter 4 describes our implemented solution and the rationale behind our design and implementation choices. Chapter 5 presents the evaluation setup for our solution and the results of the evaluation. Chapter 6 outlines possible threats to the validity of this research. Finally, Chapter 7 summarises our findings and discusses possible directions of future work.

Chapter 2

Background & Related work

2.1 Background

2.1.1 Log format

Log entries found in event logs or functional traces are produced by placing logging statements in specific locations in the source code. This process of modifying the source code with the inclusion of logging statements is often referred to as instrumentation [18]. Depending on the technologies behind the software system or component and the technology used for instrumenting the code the logged entries can differ in their contents, verbosity and format. Regardless of these differences, an entry is generally composed of fields. One field that is frequently present in logged entries is the **Timestamp** field, others may vary depending on the context but may include, category of the event, message and the location of the logging statement which produced the entry [19, 20]. These fields can be separated by different delimiters (e.g., white space, semi-colon, comma, etc.) [19, 20]. Figure 2.2 presents an example log entry.

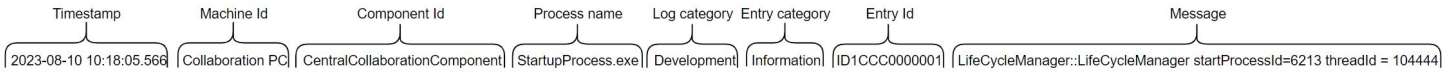

Figure 2.1: Log entry format

The contents of some fields are static, in the sense that they do not change from one occurrence of the entry to another. ComponentId can be thought of as such a field, as it should only change if the logging statement is moved to another component. Fields, however, can also contain dynamic values. For example, in the Message field in Figure 2.1, the "startProcessId" and the "threadId" are dynamic as their values may change depending on the execution of the system. This is illustrated in Figure 2.2

Figure 2.2: Entry field format

2.2 Related work

In the following section we discuss related work categories by methodology. Table 2.1 contains a summary of all discussed works.

2.2.1 Pattern-based tools and techniques

Multiple works have leveraged the use of patterns for supporting log analysis. The most closely related systems are Nfer [11] and TeSSLa [10]. Both of these systems enable practitioners to define specifications and abstract the contents of logs based on the specifications. Both the specification and the results produced by Nfer are based on temporal intervals that can carry data. Therefore, users need to specify the temporal intervals to which Nfer should abstract, by writing rules. While the syntax for these rules is basic, users need to learn it in order to use Nfer, as there is no visual interface for it. TeSSLa is a temporal stream-based specification language for Stream Runtime Verification (SRV). The language is designed to monitor a specific class of real-time signals. Both of these systems require extra efforts from practitioners in learning how to write the specifications as well as writing and maintaining the specification files which can consist of tens or hundreds of lines of text. The system we present in this study provides an interactive and visual way to define to search for such patterns. Furthermore, both systems require the traces to be provided in a very specific log format which is not standard. On the contrary, our approach accepts logs in JSON format[1], a widely used format in the field of software engineering.

2.2.2 Visual tools and techniques

QBE[21] is a query language for querying relational data that provides a graphical user interface (GUI) to facilitate the creation of queries. In QBE, users are able to construct queries by specifying example tables. Our proposed solution enables users to query structures in logs in a similar way - by defining the specific structure from an example (i.e., entries in the analysed log). This similarity between our solution and QBE stems from the goal of both works to provide easy-to-learn and use functionality. Nonetheless, there are several important differences. While a log can be represented as a relational table, QBE queries cannot used to query structures as the queries can only express conditions based on equality or inequality. Hence, it is not possible to specify conditions on individual rows of the table. (e.g., construct an example table in which different conditions apply to different rows). Moreover, as logs consist of sequential data, the ordering of entries is integral. Relational databases however may not be able to capture the full richness of such sequential data.

Previous studies have investigated analysing logs through the use of visualisations and interactivity [13, 12, 9, 22]. Cappers et al.[9] describe Eventpad, as a system for visual analytics that provides users with visuals to explore existing and discover new patterns in a collection of event sequences (i.e., logs). Eventpad provides an interface consisting of glyphs (i.e., grey squares) to represent log entries. Eventpad users can specify rules, based on regular expressions, to explore particular patterns in the collection of logs. These rules can be applied for the discovery of new patterns, the verification of certain existing patterns and/or the removal of obsolete ones. Additionally, EventPad offers several views that can be used for the comparison and/or alignment of the different sequences. but the tool is meant for analyzing a collection of logs, not a single log, which is the common scenario for software engineers investigating bugs. Several other tools such as EventFlow [13] and (s|qu)eries[12] provide interactive visualisations similar to those in Eventpad, however, both of them have been evaluated only on specialised event data

[1] https://www.json.org/json-en.html

(patient records and shopping website log) and their usability on software produced event logs and functional traces is not investigated. and At the time of writing Eventpad, Eventflow and (s|qu)eries are not publicly available.

Wininger et al. [8] present an extension to the open-source (OS) tool TraceCompass[2] allowing developers to create a custom analysis of an execution trace. Users of the extension can define a state machine in a domain specific language and visualise the result of the stateful analysis in custom Gantt charts or XY charts. While the extension provides flexibility by supporting a range of log formats, it requires users to write extensive specifications that contain the state machine definitions and details of the desired visualisations. In our work, we prioritise the ease of use and convenience of users so that they are not deterred by excessive efforts.

Beschastnikh et al. [14] describe ShiViz, a visualisation tool that helps developers debug distributed systems. As a distributed system consists of multiple nodes (i.e., computers) that may operate concurrently and interact with others, event logs and traces produced by these systems tend to be large and overwhelming to programmers. ShiViz provides a visualisation consisting of a log view and an interactive sequence diagram in which developers can easily follow sequences of events through the different nodes and distinguish between the events from the different nodes. Furthermore, the sequence diagram lets developers easily hide entire nodes and events occurring on them. Furthermore, ShiViz provides developers with the ability to search for specific patterns by drawing sequence diagrams in which they could specify a particular sequence of events occurring between nodes with node names constructed with regular expressions.

Other studies present techniques and tools for automatically visualising the contents of logs [23, 24, 15]. Sabalan, presented by Alimadadi et al. [15] is the most closely related tool to our solution as it explores the presence of patterns in logs. However, the purpose of the tool is to enhance software comprehension. Sabalan infers "motifs" (i.e., patterns) from traces by applying a generic technique inspired by bioinformatics which captures patterns in a hierarchical model. This captured model is then output in a visualisation which allows users to view the patterns and interact with them. Sabalan is implemented in JavaScript and can be utilised only to analyse software written in JavaScript. Our implementation differs from Sabalan in both intended purpose and scope. Specifically, our goal is to facilitate the analysis of software bugs through user-defined patterns, not to improve software comprehension through automatically inferred ones. Moreover, our solution does not require source code to be instrumented and can be used with software written in different languages.

[2] https://eclipse.dev/tracecompass/

Table 2.1: Summary of related work

Authors	Year	Tool / Technique	Purpose
Jerding et al. [23]	1997	Interaction Scenario Visualizer (ISVis)	Reveal interactions taking place during program execution
S Johnson [21]	1999	Query-by-Example (QBE)	query language for relation data for constructing queries through a GUI
Fails et al. [22]	2006	PatternFinder	Provide a visual interface for querying temporal patterns within multivariate and categorical data sets.
Wongsuphasawat et al. [25]	2011	LifeFlow	Provide interactive overview of event sequences
Monroe et al. [13]	2012	EventFlow	Explore point and interval event patterns in event sequences
Zgraggen et al. [12]	2015	(s|q)uerries	Query and explore event sequences through regular expressions
Kauffman et al. [11]	2016	nfer	Provide user defined event stream abstractions.
Cappers et al. [9]	2017	EventPad	Event sequence exploration
Wininger et al. [8]	2017	Trace Compass	Provide user defined stateful analysis of execution traces
Alimadadi et al. [15]	2018	Sabalan	Infer hierarchical motifs (i.e., patterns) from execution traces
Convent et al. [10]	2018	TeSSLa	Provide a temporal stream-based specification language for stream runtime verification
Jonk et al. [24]	2019	measurement-based approach	Propriety tool for inferring Time Message Sequence Charts (TMSCs) from execution traces of large-scale component-based software systems.
Beschastnikh et al. [14]	2020	ShiViz	Visualise distributed systems executions
Zhou et al. [26]	2021	debugging methodology	Debugging methodology for microservice systems combined with visualisations created with ShiViz

Chapter 3

Interview Study

3.1 Methodology

To answer the first research question (Section 1.2) and to refine all our research questions we conducted an exploratory interview study. Due to the limited amount of literature relevant to visualisation and interaction with structures and the lack of documented information on the topic at Philips IGTS, this study was based on grounded theory [27], a method used to generate theory rather than to validate or test existing ones. Using grounded theory, theories are generated inductively from data such as unstructured text (e.g., field notes, interview transcripts, etc.), structured text, as well as images and diagrams. In our interview study, we conducted interviews with 8 employees from several departments within Philips IGTS. The transcriptions from these interviews were analysed using open and categorical coding.

The rest of this chapter is structured in the following way. Section 3.1.1 describes the data collection process. The data analysis is presented in Section 3.1.2. Then Section 3.2 contains our findings.

3.1.1 Data collection

We collected our data through the use of semi-structured interviews, a frequently used data collection technique in the field of software engineering [28]. In semi-structured interviews, questions are prepared prior to the interviews and posed as written, however, additional questions can be asked during the interviews, if needed. This method of data collection provided us with the needed flexibility to obtain both contextual information about the log analysis performed by software engineers at Philips IGTS, as well as specific information related to our research questions. We conducted a pilot interview with a software engineer which took 50 minutes. This interview resulted in the removal of one question from our initial pool, as we deemed it was too difficult to answer. The interview questions (Table 3.1) were approved by the Ethics Review Board of Eindhoven Technical University (TU/e) (Ref: ERB2023MCS9) and Philips IGTS management. Furthermore, we followed the ethical guidelines described by Strandberg [29]. During the interviews, Microsoft Teams was used to record the conversation and obtain initial transcripts.

Interview question decisions

In the process of designing the interview questions, we chose to avoid using the word "structure" explicitly to avoid biasing the participants. As the aim of the study was exploratory, instead, we decided to include questions focused on two aspects of log analysis - the information used during the analysis as well as the workflow. By understanding which parts of the logs (i.e., timestamps, textual description, etc.) provide the most value during analysis and what actions are performed, we could observe whether participants utilise structures, explicitly or implicitly. However, if an interviewee mentioned "structure" or related terms such as "pattern" and "interval", we posed follow-up questions by using the mentioned term. Table 3.1 presents the final selection of questions used during the interviews.

Participants

In order to select the participants, convenience sampling [30], a form of non-probability sampling, was applied. A representative of Philips IGTS management provided us with a list of twenty-three employees from multiple departments, occupying different positions who work with event logs and traces. Because our study was based on grounded theory, we selected and approached participants from this list iteratively (i.e., selecting and contacting an employee based on the previously conducted interview) in an attempt to apply theoretical sampling (i.e., the identification of further sources of data, based on gaps in the emerging theory). This approach could be applied to a certain degree as our data sources were predetermined by the initial list of employees. Regardless, it proved advantageous by allowing us to select participants based on the gaps in our theory. Each participant was contacted via an introductory email which provided the participant with a short description of the research project and a call to action: a link to a form containing a quick way to indicate interest in interview participation, as well as more detailed information about the data collection and analysis procedure. We interviewed eight participants before deciding that theoretical saturation was reached. At that time, we had contacted twelve employees of which one was not interested and three did not reply. Background information on the eight participants can be found in Table 3.2.

3.1.2 Data analysis

Coding

Each interview transcript was refined and shared with the corresponding participant to provide the opportunity for corrections or addition/removal of information. Only two of the participants provided modifications to the original transcripts. Both included additional clarifications. We did not obtain responses from the other participants. Each transcript was anonymised and coded [31] using the qualitative data analysis software ATLAS.ti[1]. The coding was done in three steps. Initially, we performed open coding by employing a combination of in-vivo codes and constructed codes [31]. We opted for in-vivo codes as our interest lay in the experiences of the participants and in-vivo codes are used in studies which prioritise the participant's voice. Emerging codes were compared and refined. Then codes were grouped into categories. Finally, the categories were organised in themes which were motivated by our research questions.

After coding each transcript we recorded the changes to the codebook - the collection of our codes [32]. The codes and categories obtained after each interview provided us with an indication of whether theoretical saturation, the point at which no new findings emerge [33], was achieved. We did not have an apriori definition of saturation. Therefore, we used a stopping

[1] https://atlasti.com/

criteria inspired by the one described by van Breukelen et al. [34], in a study which employed interviews in a similar fashion. Our stopping criteria consisted of two additional interviews, and a 95% threshold, meaning that at any point, if the last two interviews had introduced 5% or less to the total number of codes, saturation would be reached. Therefore, after the eight interviews, we deemed that saturation was reached, because the seventh and eighth interviews combined introduced 3% of the total number of codes in the codebook.

Comparison with Artificial Intelligence (AI) Coding

During coding the meaning behind the words of an interviewee is interpreted by the researcher. Thus, there is a risk of misinterpretation [35]. One commonly used method for lessening this risk is member checking [36]. We considered using this validation method by sending the codes with their corresponding explanations to participants. However, given the required efforts to do so, the availability of our participants and the low response rate to the shared transcripts we opted not to use member checking. Instead, in an attempt to mitigate possible misinterpretations, we decided to apply AI coding, a novel feature of ATLAS.ti, to one of our transcripts and compare our codes with the AI-generated ones. After coding the selected transcript we obtained 142 codes, while the AI produced 73 codes. While some of the AI codes conveyed similar interpretations to our own, the AI codes were more general. For example, the following quote:

> *"Of course, normally when they indicate that there was something wrong with the system, there is always a time, for example, "it started around three" and that's a good starting point to search for errors and any error from that point on, that is an indication of something went wrong. It might not be the root cause of the issue you're looking at, but it can be a clue. Then I go and see whether is some logical log that could be the root cause of that description. Then I start using the software."* (P1)

resulted in the AI code "Process improvement: Problem-solving strategies" while our corresponding code is "Investigation Scope: using timestamp as a starting point". While both codes are applicable, our code is aligned with one of our topics of interest - information used during log analysis. In contrast, when AI codes are generated there is no specific context or research goal. Therefore, the resulting codes are generic. As many of the AI codes were not aligned with our interests, we found AI coding to be disadvantageous for our analysis. Consequently, we coded manually all of the other interviews. However, we altered three existing codes as we found the AI-generated ones more appropriate.

In the following section, we present our explanation of the derived codes by referring to quotes by our interviewees.

Table 3.1: Interview questions.

Background questions

1. What is your job title?
2. What kind of systems is your unit responsible for?
3. What is your role and responsibility within your team as X (i.e., current job title)?

Questions about logs and traces (RQ1)

4. For what purpose do you use event logs and traces?
5. How many different types of event logs do you work with?
6. How many different types of traces do you work with?
7. What is the difference between the logs and the traces?
8. What information within the logs is most relevant for your purposes?
9. What information within the traces is most relevant for your purposes?

Questions about workflow (RQ2)

10. How often do you analyse logs for your purposes?
11. How often do you analyse traces for your purposes?
12. Can you describe your workflow when analysing a log for X (i.e., purposes mentioned by the participant)?
13. What challenges do you face when analysing logs?
14. How do you cope with these challenges?
15. Can you describe your workflow when analysing a trace for X?
16. What challenges do you face when analysing traces?
17. How do you cope with these challenges?

Ending

18. Having discussed some topics about log and trace analysis, would you like to add some thoughts?
19. How many years of experience do you have as a X(i.e., current job title)?
20. What is your educational background?
21. Do you have any questions?

Table 3.2: Background information on participants.

Department ID	Participant ID	Educational Background	Position	Years of Experience
1	P1	UCS	Software Developer	22
	P2	UCS	Product Owner	18
	P3	UCS	Software Developer	6
2	P4	GCS	Software Developer	2
	P5	UCS	Senior Software Developer	20
3	P6	GOth	System engineer	6
	P7	GOth	Chief engineer	42
4	P8	GCS	Software Designer	22

GCS: graduate degree in computer science UCS: undergraduate degree in computer science, GOth: graduate degree in other science subjects (e.g.. physics)

3.2 Results

3.2.1 Observed types of Structures (RQ1)

We observed that the concept of a structure is present in the minds of our participants and is utilised during log analysis. Some participants have extensive experience with a specific part of the code base and are aware of most of the processes and the information logged during their execution. They are conscious of an implicit high-level structure occurring in a log: "*So, normally, there is a structure in the trace files and when I know that something is going wrong in this part and this [other part] it is not interesting for me*" (P1). The recognition of such a high-level structure is beneficial for developers when navigating the file and establishing the scope of a fault investigation: "*There is a structure. It's a logical flow, so you get structure and you will recognise it after a while that some parts aren't interesting [during investigation].*" (P1). Another type of structure which we observed more frequently was a sequence of entries logged during the execution of a particular process, commonly referred to as a *flow* or a *sequence* by the interviewees. While both event logs and traces can consist of thousands of such sequences of entries, developers are aware of specific structures, the knowledge of which they utilise when analysing event logs and traces: "*The flow will always be the same across processes, except if there's an issue. You can really use that flow...*" (P3).

This knowledge proves to be beneficial to software engineers in several ways:

Defining an investigation scope using structures

Many interviewees often use event logs just as a starting point in their investigation, as the scope of the information there covers an entire (sub)system containing multiple components: "*So from the event log you get the high-level picture...Then you go to one level below that's the traces and then you look at what are the [specific details] that information you can get from the unit specific trace files.*" (P4). In the event log they localise the problematic component, generally employing simple strategies such as referring to a timestamp or performing a quick keyword search. Then they thoroughly analyse the traces of that component: "*If you really have to know the details, then you have to look at the trace files*" (P1). This top-down approach concurs with the findings in a study about the use of logs performed by Yang et al. [5]. Because traces contain information about a single component, they are much more detailed. As a result, engineers struggle to navigate through large amounts of irrelevant to the particular case information: "*...also that's one of the problems. We have just too much tracing and even for some process we are not actively developing anymore so we don't want to remove them*" (P4). Hence, finding the point of failure is a challenge: "*The other challenge is finding out where is the issue. If you have a clear tracing and a good tracing and you can find your way quickly and you can have it, for example, in an hour you have a decision, but sometimes it takes days to go through the tracing, going back to the code, checking again, asking precisely [about] this scenario, when it happens and then the challenge here is to reduce the time, that overhead for finding out the issues.*" (P5). To cope with this challenge engineers utilise their knowledge of a structure: "*use the flow of the trace because you know at certain steps, this and this should happen and so on.*" (P3). Thus, they are able to restrict the scope of their investigation to certain files or certain parts of the files. Regardless this scope could still contain hundreds or thousands of entries.

Building stories using structures

A large amount of the knowledge about the logged entries lives in the minds of the developers [16]. This knowledge lets them pinpoint which entries could be relevant to an investigation and

should be further inspected. Developers actively seek ways to separate these relevant entries from the rest. By doing so they are able to build stories and form mental models of what occurred in the software that produced the log: "*then I can also filter on that interface because I know I'm investigating some problem with it. Then I can also see the interface calls which are happening only for that interface... then you can easily form that mental model from the traces. For example, first, it was in moving and then it went to stopped. Then whether the position was reached or not, all that information.*" (P5). While in some cases this separation can be achieved by a keyword search, in others, the contents of the interesting entries are heterogeneous and cannot be simply searched for. In such cases, the approach participants take relies heavily on the availability of tooling.

Engineers who do not have access to, or choose not to use tooling develop different strategies such as manually inserting bookmarks in the files and then searching for those bookmarks: "*Yeah, let's say if it is heterogeneous, right? If it is, if I'm only looking at "ABC", then I can simply search for that. But let's say if it is a combination of "ABC" with something else. Then I cannot find that. Then I can actually add my initials to those lines. This one [entry] I need from "ABD". And OK, I need something else, some other function, I need to add from something. I'll just mark this [entry] and then I only get that list [containing all the marked entries].*" (P5). Another common strategy is to open several instances of the same file in a text editor, focused on different entries: "*you can imagine one is at the top of the file and the second one I'm interested is at the bottom. Sometimes I also put two files next to each other because that's the old trick. Then I can see this process put next to each other*" (P5). Such strategies, however, require several manual steps to be taken (e.g., opening files, scrolling to specific parts, etc.) and can be inconvenient: "*...this is still very cumbersome but these are all the tricks we have to do*" (P5)

Participants who have access to tooling rely on the use of features that let them pin entries which can then be seen in a separate view or filtering features that let them define custom filters on the fields of the entries: "*So you can really pin and if I pin this again, so you can you can build a story from here*" (P3), *"so to categorise them I can right here ... filter the steps of the happy flow*" (P4), "*...there is a pinning functionality. Yeah, we use it because you have a lot of entries and at a certain moment you will have to filter your entries and by pinning the filtered entries then you have reduced view of the tracing and you can follow better the relevant entries. For example, a process which is taking too long to start. You will start by the start of the main and the constructor of the instances within that process and you will filter on that. Based on the construction of all instances during the startup, you can check how long this construction takes and where is the delay for example.*" (P8). However, many of these tools are self-made and contain many hard-coded parameters which limits their use to a single participant or a team of engineers.

With or without tools, developers generally constrict the scope of their investigation to multiple logged entries: "*...maybe max ten in worst case. Otherwise, if I'm pinning many entries then it doesn't make sense. Maybe then I'm not doing the right investigation also.*" (P4), "*...it usually this is four max, maybe five [entries]*" (P5) which describe actions the system took when a fault occurred. We consider these entries to be representative of a structure.

Comparison of logs through structures

Some software faults occur intermittently: "*The system behaves OK and at certain time the issue may happen or it may not happen, and sometimes you have a very intermittent issue which happens once in a thousand times and sometimes it happens one in ten times.*" (P8). Such bugs cannot be easily reproduced, which is why developers generally use another approach: "*Mostly when we have an intermittent issue, for example, you will compare the good or the right tracing,*

and the tracing when it [the issue] happens." (P8). By comparing a log produced during the occurrence of an (intermittent) fault with one representative of the normal execution, developers expect to find the origin of the fault. However, performing such a comparison is not trivial as there can be a large number of differences between two logs describing different executions of the same system or component. Not only can the timestamps of all the entries differ, but so can the order of entries due to concurrency in the software, and some of the contents (e.g., different thread and process identifiers, etc.). Therefore, a common way used to perform this comparison is to open the logs side-by-side and manually inspect them for differences and anchor points (i.e., entries by which to align the contents of the logs). Engineers utilise their knowledge of specific structures by implicitly using the structures as anchor points: "*if you don't have an accurate enough explanation of what went wrong, then the only thing you can do is have a similar scenario on the system that does work, put the trace files together and just look at the differences. If everything was synchronous, then everything would be very easy, because those steps could be compared, but it's in parallel and its not easy to put two trace files side by side and just compare them. It's not like the trace statements have followed in the exact same way. Yeah, it's multi-threaded so trace statements are not really aligned and of course there are some transitions you can check - "okay, now I have this transition and it should be similar to that one" and then you can align those traces again a little bit, but line by line putting them next to each other, that's impossible. So you really have to look at state transitions then, to see where they behave similarly. Those are the synchronisation points, transitions towards another state or an action towards the hardware, a user action. That is where you can synchronise those trace files. There are some clusters of around twenty trace statements from network, which should happen in all trace files, so those combinations of lines can also be used to find similarities.*" (P1).

RQ1 Summary: We observed two types of structures that experienced engineers become aware of. The first type is a high-level structure of a log. The knowledge of such a structure is used by engineers to constrict the scope of a bug investigation to a log or a certain part of a log. The second type consists of a sequence of entries often related to a specific process and is often referred to as a "flow" of a process or a "sequence". This knowledge of such structures is more prevalent than that of the other type. Engineers often rely on this knowledge when investigating bugs, to avoid investigating irrelevant entries, forming mental models and comparing logs.

3.2.2 Tool features supporting the use of structure(RQ2)

Observations on the current availability and use of tooling

As previously stated, engineers commonly use of self-made tools and text editors during log analysis [5]. Before presenting possible features supporting the use of structures, we discuss several observations on the current state of availability and use of tooling. Most often engineers have access to a self-made tool used to facilitate the analysis of functional traces or event logs. However, these tools generally are hardcoded to a specific format of logs and cannot process both event logs and traces due to the difference in the file formats. Therefore, they are often used only by groups of engineers working together in a team or in some cases, individuals. Regardless of the different formats, the tools for both event logs and traces often offer overlapping features such as: parsing and displaying the contents in a more readable way, searching for keywords, highlighting entries based on Boolean statements and filtering based on keyword or Boolean statement. Nevertheless, not all tools implement all mentioned features. We observed a case in which different teams were using custom tools offering different features and engineers from both teams desired features implemented in the tool of the other team.

Engineers who develop or use such tools develop a certain loyalty towards them:"*that's how I use the tool at least. So I use this tool a lot. I use this functionality as well*" (P4), "*I just said I'm kind of biased, so I like this the most of the functionality*" (P6), "*I use only the [tool] because I used to use it and I like the tool or I am more familiar with it.*" (P8). Such loyalty can harm the workflow of engineers. For example, such tools are often used for long periods after their development and improvement have ceased. However, the enhancement of such tools rarely happens because they have been developed and maintained by individuals. In such cases, engineers using the tools adjust their workflow to cope with certain disadvantages and lack of features instead of improving the tools or searching for new ones.

Other practitioners opt not to use tools as they require certain prerequisites or are difficult to use: "*I use just a text editor. I don't have a special tool. We do have special tools that we've developed ourselves, but normally it's so hard to set up [the tool] that I immediately go for the text editor and immediately start searching. It's mostly faster than using a complex tool that you have to configure and set up and fill a lot of things in...I think that every software developer is lazy, so try to grab something that's the easiest.*" (P1), "*I know there is one tooling called [the name of the tool]. Because that tooling was developed later. I think later, that's already 3-4 years back. But before that, we didn't have any tooling, we just got used to it and then also the tooling expects the traces to be in a particular format. you know, I already showed you right? Because we have different ways of tracing. then it's also a lot of effort to make it the same.*" (P5)

Given our observations, we make the two choices for the tool features presented in the following section. First, the features will be implemented in Tracy (Section 4.1), an open-source platform being developed in the Accelerando project to investigate how log analysis can be supported. By implementing our solution within Tracy we ensure that it can be further investigated, developed and improved. Second, we prioritise the ease of use of the features to avoid discouraging software engineers from using our solution.

Feature for encoding domain knowledge during analysis

Although developers utilise their awareness of structures intuitively during analysis, they lack tooling with which to use this knowledge explicitly. We believe a feature that enables engineers to define relevant to the analysis structures and highlights their presence or lack thereof may be advantageous to developers. Such a feature could possibly address one significant challenge of log analysis, reducing the large amounts of logged entries needed to be checked by highlighting relevant structures through all the irrelevant entries, also referred to as "noise" by engineers: "*Yeah and then indeed [the challenge is] finding those right ones [logged entries] and you can get to them as soon as possible then you're already there. All the rest is noise...*". Flexibility is an important aspect of such a tool so that engineers are able to define various structures. Structures could vary in number of entries as well as specific entry fields. For example, if a specific structure reoccurs multiple times in a log, the timestamps of the entries in the structure will be different. Therefore, the timestamp field is not relevant to the structure.

Feature for sharing domain knowledge

While the domain knowledge of software engineers about logs increases with experience, this knowledge is finite. Furthermore, as software constantly evolves, engineers may have to work with systems and components with which they are inexperienced. In such cases they struggle with analysing logs due to lack of knowledge of the source code and the logged information: "*Yeah, it depends on the experience, it depends on which part or which component, which part of the code. For example, I'm working for a long time now with some code which I never touched and I don't know what should be the sequence there and what would basically help me. And also you*

can't remember all the things which sometimes you know about it and you remember or you can predict what happened there. Mostly when you have interface calls from one component to another component." (P8), "*That's for me the main challenge. Finding, analysing the logging, getting the flow right in an unknown unit. Getting the flow without having any input, like if you're working with someone that has experience, of course he will say maybe check this first and then from there you can build up. But if you have no clue about what's happening, you're even not sure if this is part of [the bug], maybe the names... it's also not clear when you just start, so you have to build up the knowledge and that building that knowledge is for me the yeah, the main challenge when starting.*" (P3). Thus, a feature providing a way to share domain knowledge of structures could alleviate this challenge. Furthermore, such a feature could improve collaboration as currently engineers often resort to copying and pasting text and taking screenshots when collaborating on investigations: "*OK, I see this PC failing. OK, what happened after that? So I can really pin that here and also share it with colleagues. It happens a lot that you want to describe or send my e-mail what will you do. Personally, I just take a screenshot of the storyline that was created like this and then send it by mail: I see this. I see this, I see this, what do you think? And then from there you can maybe you say, OK, let's have a look*" (P3), "*So once I finish with the investigation, I have to capture my findings or observation there and propose, OK, how much is time, whether this is our unit, who has to fix or whether it should be forwarded to some other units. So in that I have to capture some information and then I have to mention what is what are my options then I'll say, OK, I looked at time this time and then there was a call and I expected the call should have come, but it has not happened. And then most of the time what I do, I take a screenshot on because. They're developers. As I said, it will be also forward if I'm forwarding it to some other subsystems like [subsystem name]. Definitely, another developer will be analysing. To save his or her time it is really handy to give the same information where I have spent time, so I generally select this and add this. But if I can select and copy-paste. That is also easier, so they also know, OK, this is the time frame [the participant] investigated and he thinks and then they can take over it easily. But now I use this screenshot and then I sometimes try to highlight and say, OK, this is the time I'm saying and I was expecting something and this call was not expected. So it takes time*"

RQ2 Summary: Based on the observations on the types of structures and their impact on log analysis, we propose two features to support the active use of the knowledge about structures during log analysis. First, a feature enabling engineers to define structures relevant to their analysis and highlight these structures in the log. Second, a feature for facilitating the sharing of such knowledge between engineers to facilitate both analysis conducted by inexperienced engineers and collaboration.

Chapter 4

Implementation

In the following chapter, we present Tracy, the platform which we extend with the features described in the previous chapter, as well as the design and implementation choices of these features.

4.1 Tracy

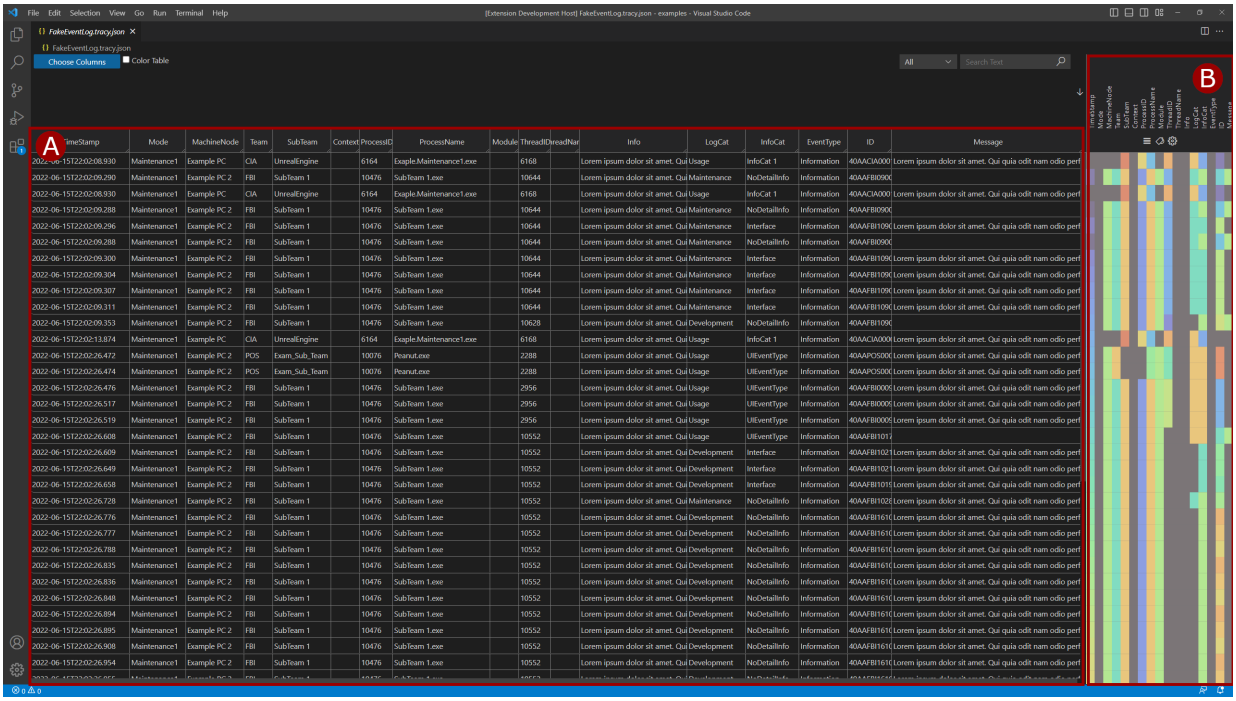

Figure 4.1: Tracy GUI A)LogView component B) Minimap component

The purpose of Tracy is to support the research of event log and trace analysis by providing a platform on which state-of-the-art features can be implemented and further investigated in

practice. At the time of writing Tracy supports the integration of state-based reasoning during log analysis, by providing users with an intuitive way to define simple state machines and visualise states throughout the analysed event log or trace. Figure 4.1 presents the graphical user interface (GUI) of Tracy.

Tracy is implemented as a Visual Studio code[1] plug-in, which lets users seamlessly transition between source code and Tracy. Furthermore, Tracy leverages web technology as it is implemented using the React[2] library to be able to provide dynamic visualisations and thus support a visual analysis.

Tracy contains two React components: *LogView* (Figure 4.1 A) and *Minimap* (Figure 4.1 B).The former component presents the event log or trace as a table in which every row is an entry in the original file and each column is a field of the recorded entries (e.g., timestamp, event id, description, etc.). Tracy requires that the entries in the input log or trace contain the same fields. As logs can contain too many entries to be properly rendered, the table is scrollable. The Minimap contains the same number of columns as the LogView. Each of the columns contains a number of coloured cells which represent the values of the cells in the LogView table. The Minimap provides an easy way for users to quickly scroll through the log and see when the values of a field change. Additionally, it can be zoomed out to provide a "bird's eye view" over the log (Figure 4.2). As previously described, Tracy provides some state-based reasoning. It does so by providing a way to specify a state machine based on the entries in the file, which results in the addition of a new column in the table. The values in this column record when there is a change in the states of the state machine. Furthermore, this functionality integrates with the Minimap, as users can quickly spot a state change in the Minimap, in a column corresponding to the one in the LogView.

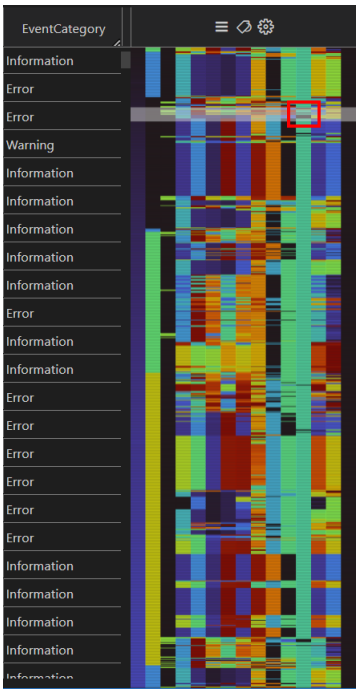

Figure 4.2: The Tracy Minimap zoomed out to provide a "bird's eye view". The red rectangle corresponds to the entries partially visible in the log to the left of the Minimap.

As we are performing our study in the context of the Accelerando project, the functionalities from the findings of our interview study Section 3.2.1 were implemented in Tracy. In this way, they can be further studied and developed when Tracy is utilised in future case studies.

4.2 Features

Due to the time constraints of the project, we opted to perform the implementation and evaluation iteratively, by implementing one feature and evaluating it (Chapter 5) before continuing with the next one. This approach would allow us to evaluate our work even in case there is insufficient time to implement all the functionalities from our interview findings. Moreover, we would be able to incorporate feedback and observations from the evaluation sessions when choosing how to proceed with the implementation work. By following this iterative approach we began with implementing a Structure Matching functionality.

[1] https://code.visualstudio.com/
[2] https://react.dev/

4.2.1 Structure Matching

As previously described in Section 3.2.1 experienced developers recognise the flow of processes in files they analyse during a bug investigation. However, due to lack of tooling often they do not benefit to the fullest from this knowledge. To solve this challenge we decided to implement a feature enabling developers to specify a structure of one or more log or trace entries and be able to view and traverse through the occurrences of this structure in the entire log or trace. We envisioned two use cases for this functionality that we call "Structure Matching". In the first use case, the user specifies a structure relevant to a specific bug investigation. That is a structure containing an entry which contains information revealing the (source of a) software bug. In the second use case, the user specifies a structure containing irrelevant entries which may reoccur many times, also referred to as noise by our interview participants. The benefit of this second use case comes from combining the Structure Matching with another (planned) functionality in Tracy allowing users to expand and collapse segments of the log or trace. With the combination of these two functionalities, users could easily collapse structures containing noise, reducing the investigation scope. The first use case may come as more intuitive, but the second use case may offer equal value to users because in general developers are able to recognise single entries or entire parts of the log which are of no value to a case (Section 3.2.1).

Design

In the process of designing the GUI of the Structure Matching feature, we used the "Eight golden rules of interface design" by Shneiderman [37] while striving to create a simple and intuitive interface. Below we split the Structure Matching functionality into three conceptual parts - defining a structure, modifying a structure and matching a structure, and describe our design choices.

Defining a structure The process of Structure Matching begins by selecting entries to be included in the structure. Users are able to select entries from the LogView component by either clicking on individual ones or shift-clicking (similarly as in most modern operating systems) to make a selection of multiple entries Figure 4.3. We allow the use of shortcuts, such as the shift-click (i.e., left mouse click while holding Shift), when possible to follow the second rule of Shneiderman [37] (i.e., allow the use of shortcuts). After selecting entries users, can define the structure by a button click. The Structure Dialogue (Figure 4.4 and Figure 4.5) is then displayed. In this dialogue, the user can see the newly defined structure and modify it. We consciously chose for this dialogue to be visible only when the user is using the functionality to avoid crowding the Tracy GUI and to stay consistent with the other functionalities of the platform which were also implemented through similar dialogues.

Modifying a structure

Adding and removing entries Entries can be added or removed to/from the structure at any moment to allow users to adapt the structure to their needs during an investigation.

Constraining the distance between entries Entries describing one process do not necessarily appear together in a log or a trace. Often these entries can be separated by tens or hundreds of others, as information from multiple components or processes can be logged at the same time. This distinction gave us an idea for allowing the user to define the distance between the entries in the structure for more flexibility of use. We opted for three values from which

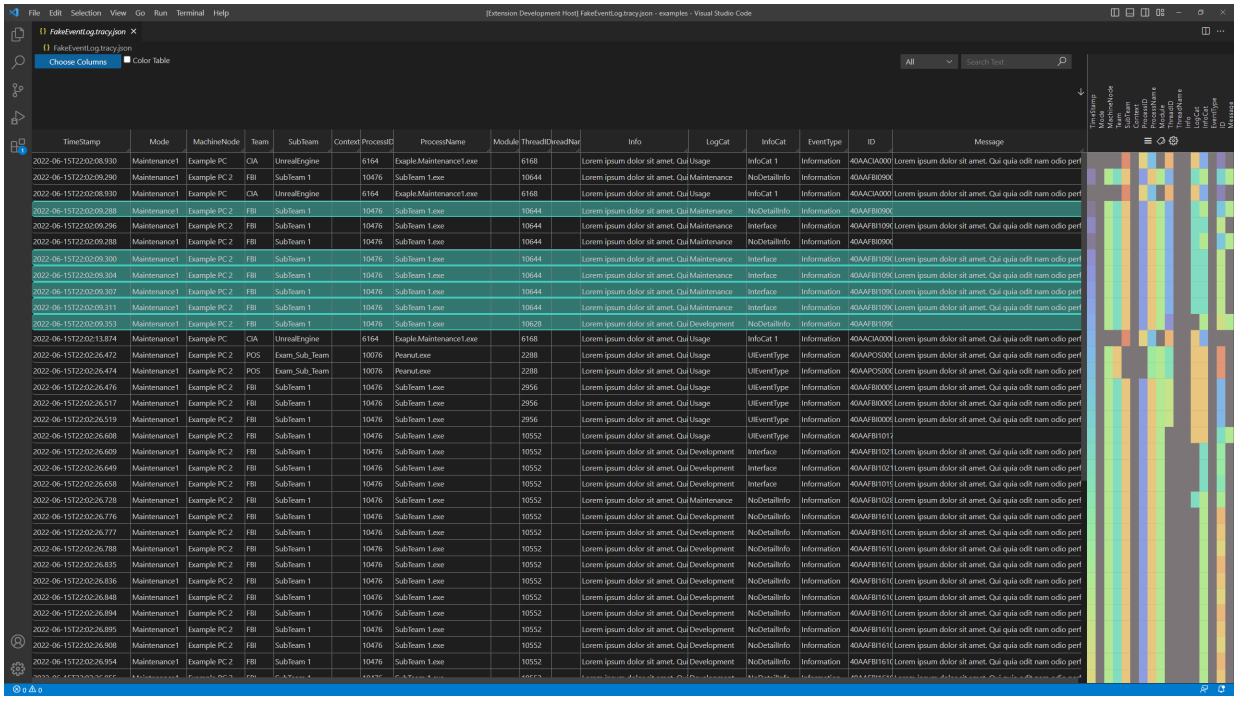

Figure 4.3: Selection of entries prior to defining a structure

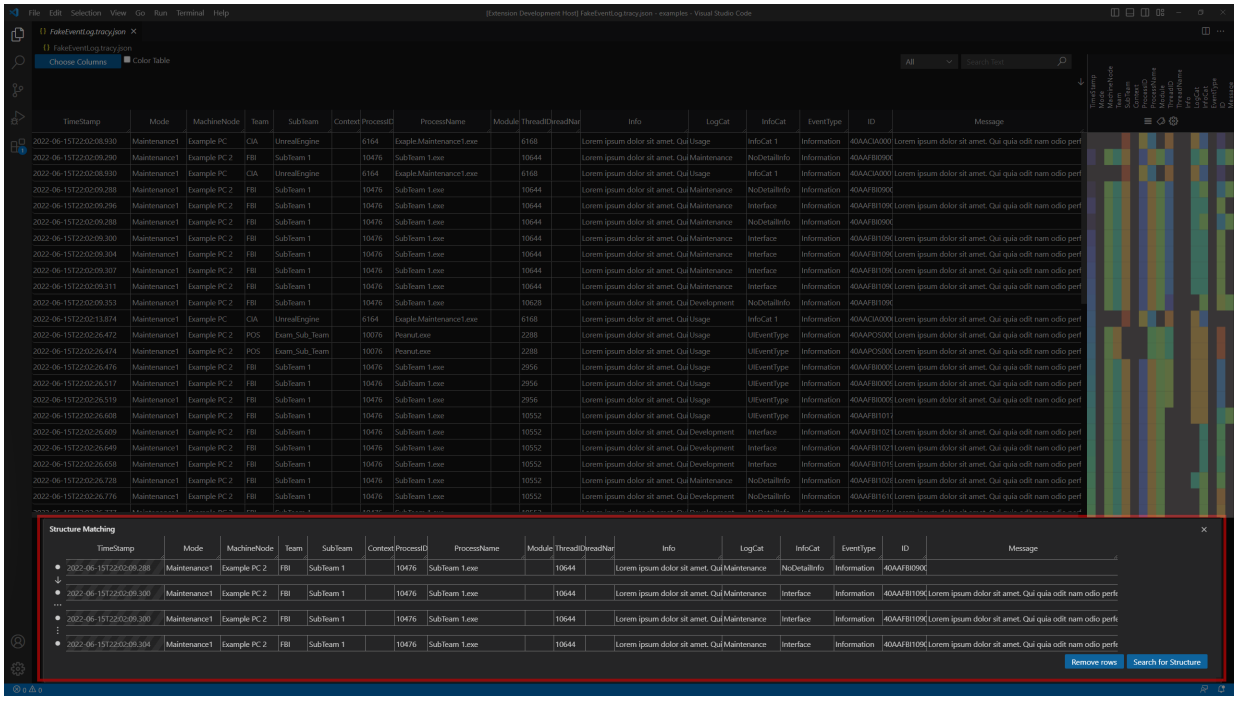

Figure 4.4: Structure Matching dialogue highlighted

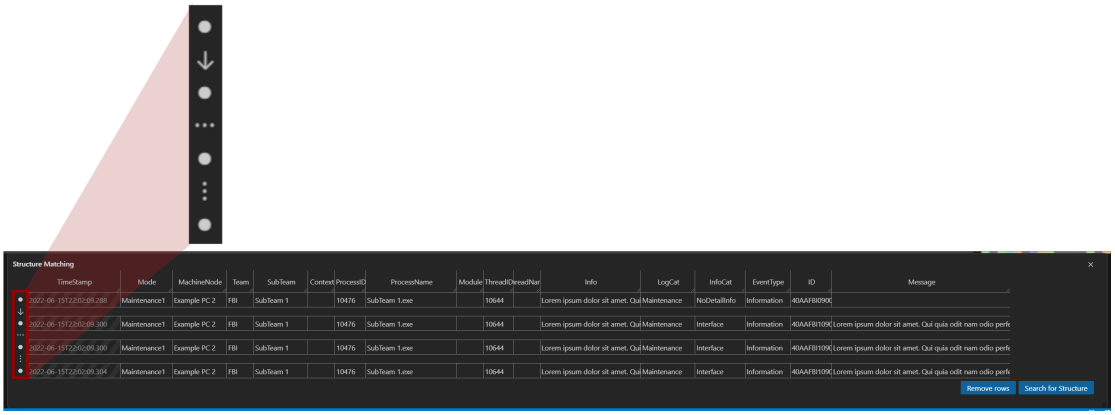

Figure 4.5: Structure Matching dialogue with highlighted constraints between entries in the structure. The constraints from top to bottom are: NONE, MIN, MAX

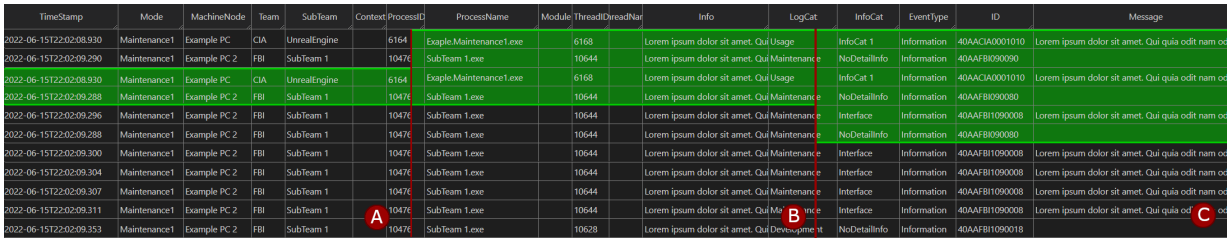

Figure 4.6: Matching with different structure link distances: A) NONE, B) MIN C) MAX

the user could choose, to which we refer to as "Structure Link distances" because these values represent the distances between entries in the structure and can be thought of as linking sections. The values for these structure link distances were chosen based on the implementation approach (Section 4.2.1). The first and most restrictive value is *none*, represented in the GUI by an arrow pointing down to indicate that the second entry should be directly below the first one. The second value, *min*, allows for zero or more entries to appear between the two structure entries, matching an occurrence with a minimal number of entries in between. This value is used as the default one and is represented as an ellipsis (i.e., three horizontal dots) to indicate the possibility of entries in between. The last value allows for zero or more entries as well but matches the occurrence with most entries in between. Figure 4.6 presents an example in which three different occurrences are found for the same structure consisting of two entries by changing the structure link distance between them. In part A of the figure, the structure link distance is set to *none*, hence the only occurrence matched is the one where the two entries are logged after each other. In part B, the structure link distance is set to *min* and the occurrence matched is the one that contains the minimal number of rows between the two entries. Finally, in part C, the occurrence with the largest number of entries in between is matched.

Modifying which entry fields are important for the structure Potentially, a structure consisting of even a single entry could occur many times in a log or a trace with the timestamp being the single difference between the occurrences. Therefore, the timestamp should not be used during the matching, unless the user is interested in a single occurrence. For this

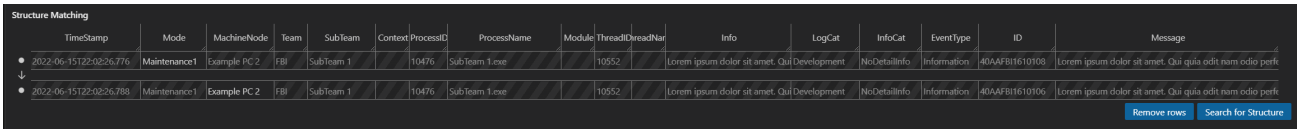

Figure 4.8: A structure consisting of two entries; The first entry should have *Maintenance1* mode, the second should have the machine node field "Example PC 2"

reason, we decided to give the possibility to toggle which fields of the structure entries should be used during matching. The user can (un)select a single field by clicking on it, or control-clicking (i.e., left mouse click while holding Ctrl) to (un)select all other fields of an entry. As this action lets the user make a change to a single field (i.e., a single cell in a row in the structure table) it was important to visually distinguish between important and unimportant fields because a single cell may be small and not easily visible. One challenging aspect of visualising this was that simply using colour (e.g., darkening an unselected cell) might not be distinguishable enough unless using drastically different colours. Instead of relying on the use of colour, we applied a principle for generating accessible images described by Katsnelson [38] - to use shapes and line textures. Therefore, we visualise unselected fields by covering the cells of these fields with diagonal stripes. Such a striped pattern is commonly used as a road marking to represent inaccessible spaces. Figure 4.8 displays a structure consisting of two entries, in which different fields are used during Structure Matching.

Matching a structure The user can match the structure visible in the Structure Dialogue at any time, with a button press (Figure 4.4). The first occurrence of the structure is then highlighted and displayed in the LogView component. Nevertheless, there could be multiple occurrences of the matched structure (e.g., one at the top of the log and one at the bottom). Therefore, a way to inform the user of the number of matches and a way to easily navigate through them is needed. For this, we chose to utilise a design similar to the one used in the search functionality of most modern browsers and integrated development environments (IDEs) (Figure 4.7). This interface is visible in the Structure Dialogue after the user has attempted to match a structure. The user is able to see the number of matches as well as the currently highlighted match. By default, the first match is highlighted in a lighter tint of green from the rest. However, when the user navigates through the matches, they all get highlighted in turn to reflect the navigation (e.g., when navigating to the second match, it will be highlighted instead) (Figure 4.9).

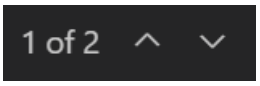

Figure 4.7: Interface for navigation through structure matches

Design and purpose validation During early stages of designing the Structure Matching feature, we presented the design of the interface as well as the purposes of the feature during a meeting between the Tracy development team and several product owners from Philips IGTS. We received much positive feedback from the product owners which displayed enthusiasm for the feature. We considered this feedback as validation towards both the design as well as the purpose of the tool.

Use of colours To keep the GUI as simple as possible, we decided to use colours and highlighting as a way to visualise the selection of entries prior to the matching and the visualisation of the results. We chose one blue colour for highlighting the selected entries, and two green

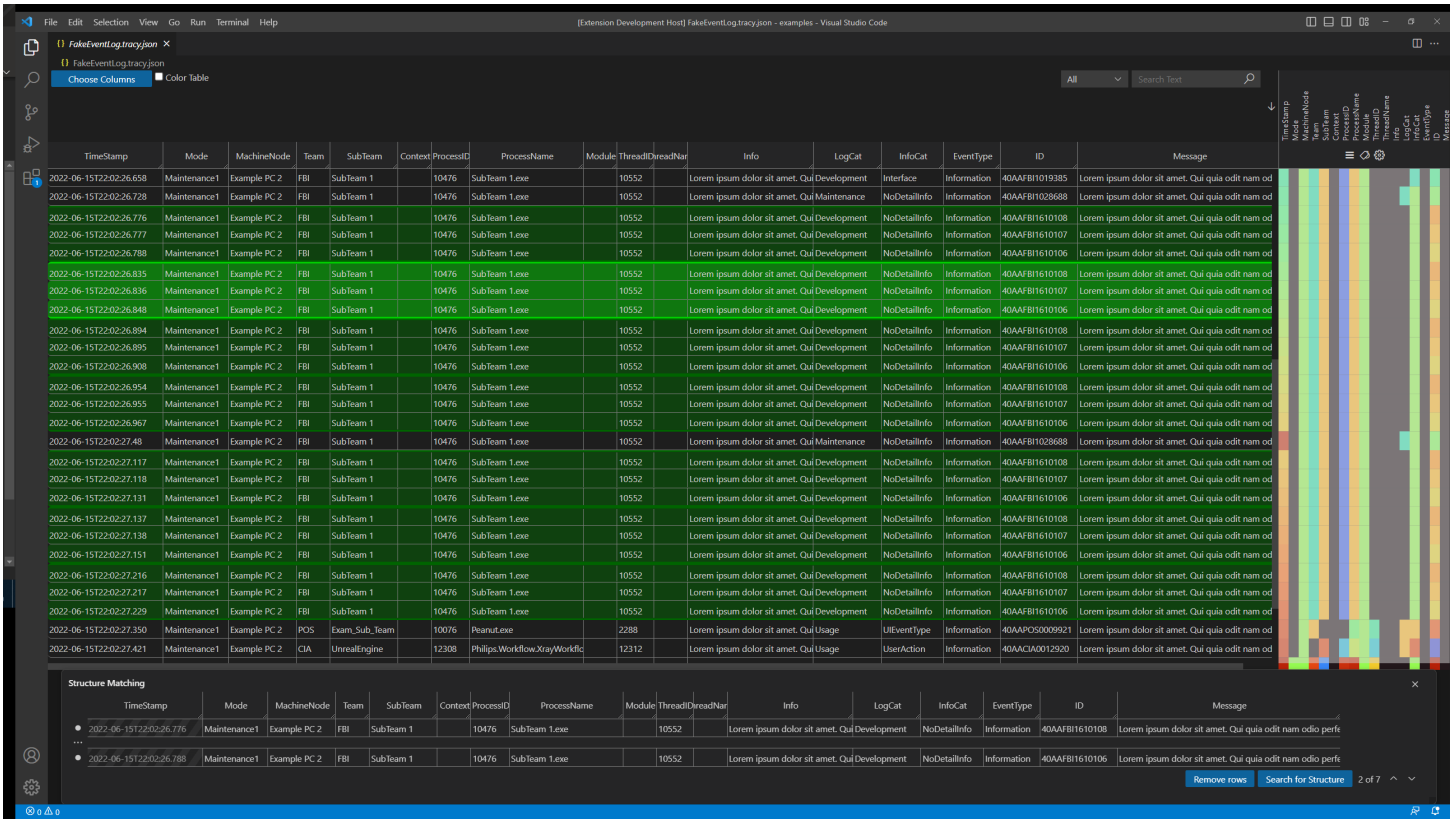

Figure 4.9: Visualisation of multiple matches. The second of the matches is currently highlighted.

colours for the matched entries. As we heavily rely on the use of colours, the choice of which to use was a significant one. Colours could have different meanings and associations for users based on religious, symbolic, biological (i.e., colour blindness) and other differences. While supporting customizability can address this issue, we decided to select an initial colour scheme which could be evaluated and changed at a later point in time. We decided to use colours which can be easily distinguished from the background colours in Tracy (and Visual Studio Code). Figure 4.10 displays the chosen colours on several Visual Studio themes. Furthermore, we prioritised colours which can be seen by people with colour blindness, in order to improve the user-friendliness of the GUI (Figure 4.11).

Blue is used to indicate a selection in Windows and in Visual Studio Code. We decided to remain consistent with this colour choice for selection, however, we use a bright blue, specifically *turquoise*. The reason for this is shades of blue with high contrast and saturation, that do not contain any red or yellow hues, such as *turquoise*, are the most visible blues for people with various types of colour blindness [38]. Similarly, we chose a bright *lime green* for representing the currently highlighted matched structure and a darker *office green* for the other matches. While, the aforementioned colours can be easily distinguished apart from each other by people with the most common types of colour blindness, in some rare types of colour blindness such as in cases of complete achromatopsia [39] (i.e., total absence of colour vision) it could be more difficult to distinguish between these colours. Nevertheless, following the principles of generating accessible images proposed by Katsnelson [38] such as checking the visuals in greyscale and using a simulator - coolors web application[3], we decided that these initial colour choices are fitting and can be further improved if needed. Figure 4.11 depicts how our chosen colours are perceived by people with different types of colour blindness.

Implementation

Use of Regular expressions To implement the Structure Matching functionality we opted to use regular expressions [40], a powerful tool for finding and manipulating patterns. We chose regular expressions for several reasons. First, while through the interview study, we were able to observe structures in the analysis of our participants, we were unable to specify how complex could structures used by developers be. As regular expressions are a powerful tool for finding and manipulating patterns, successfully employed in related state-of-the-art tools [12, 9, 14, 16], we decided to use them in the initial implementation. There exist several libraries which could provide an alternative to the use of regular expressions, such as APG-Js[4] a library providing the use of Augmented Backus–Naur Form (ABNF) [41] pattern syntax or a PEG.js[5] a library generating parsers for the Parsing Expression Grammar (PEG) formalism [42]. However, these libraries are developed by small groups of developers, and contrary to regular expressions, are not widely used. In order to use regular expressions in our solution, we utilise the standard JavaScript implementation of regular expressions[6].

4.2.2 Wildcards

After evaluating the Structure Matching feature (Section 5.2.1), we opted to improve it by introducing Wildcards. Wildcards allow users to abstract away business data and technical data commonly found in various fields of logged entries from the definition of structures. To illustrate the purpose of wildcards we consider the following example.

[3] https://coolors.co/fffffff
[4] https://github.com/ldthomas/apg-js
[5] https://pegjs.org/
[6] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp

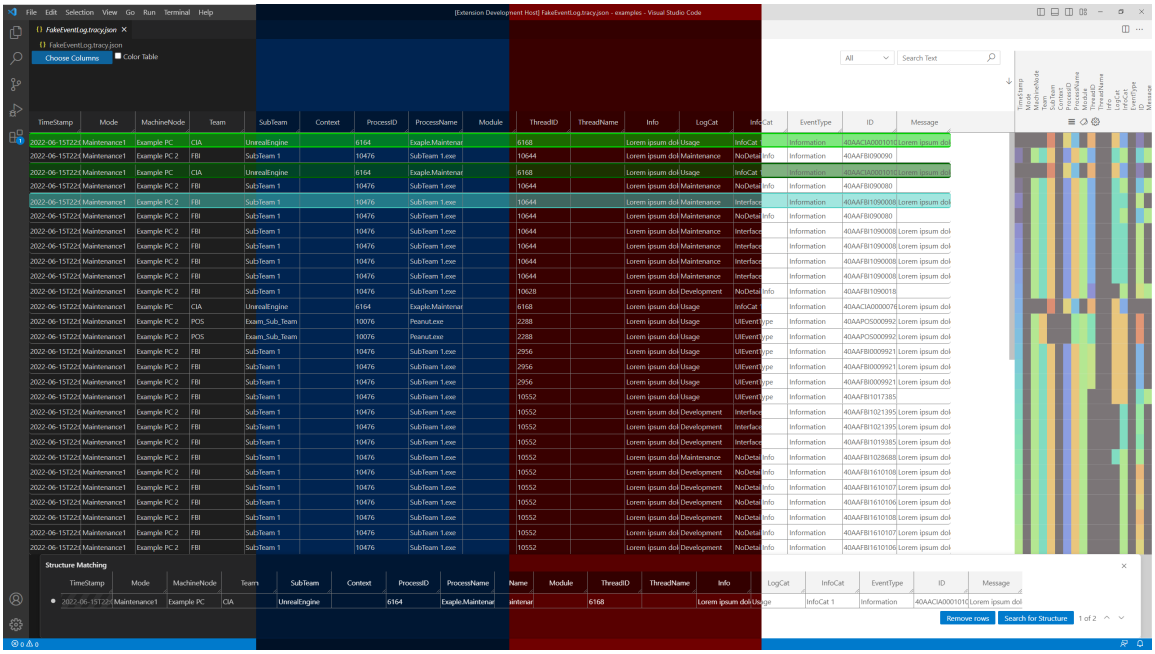

Figure 4.10: Structure Matching related colours on different Visual Studio code themes

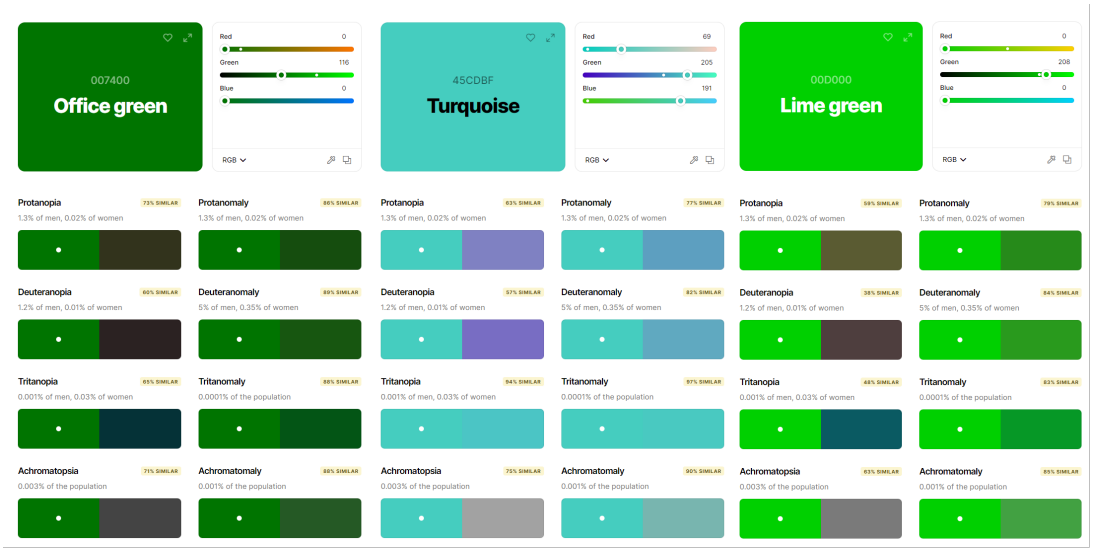

Figure 4.11: Colour values used for Structure Matching and how they are perceived by people with different types of colour blindness. Adapted from the coolors web application.

A user using the Structure Matching functionality is interested in verifying that software components have properly initialised, therefore he defines a structure describing the initialisation of a component. Such a structure definition is presented in Figure 4.12. A single occurrence will be displayed to the user upon matching the structure while there might be many such structures logged for different components. Other structures are ignored because the structure definition explicitly contains the name of the component "Component-A" while other structures contain the names of other components. In this example, the technical data (i.e., the name of the component) must be abstracted in order for the Structure Matching to capture all relevant structures. Wildcards aim to address this issue by providing an easy to use abstraction.

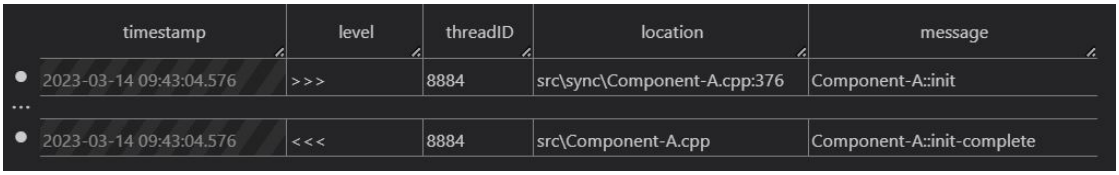

Figure 4.12: structure definition for a structure containing two entries depicting the initialisation of a component

Design

We considered several factors when designing Wildcards. First, multiple entries in a structure and their corresponding fields can contain business or technical data. Thus, the feature must support the use of multiple wildcards in a structure. Furthermore, the addition of wildcards must be streamlined. Second, as the purpose of the wildcard feature is to enhance the Structure Matching, the implementation of the wildcard must be coherent with that of the Structure Matching. Furthermore, the interface of the feature must be coherent with the previously implemented interface as per the first rule of Shneiderman [37] (i.e., strive for consistency). Finally, multiple log entries or fields of an entry can contain the same value. In the aforementioned example (Figure 4.12), the "message" field of both entries contains the name of the same component. Therefore, when abstracting the name of the component, any value should be possible, however, the value should be the same in both places. Therefore, the wildcard feature must be capable of handling such a case.

Our approach to designing the interaction with wildcards is inspired by the simple operations of copying and pasting text through the use of a context menu often referred to as a "right-click menu". These operations are simple and commonly used within text editing software. Therefore, we believe that this approach would feel intuitive to users. Furthermore, such interaction allows users to precisely and easily select text from the structure entries and through a context menu wildcards can be created or removed.

One challenging aspect of this feature was visualising the wildcards. Unknown characters are represented differently in various software contexts. In the field of databases, the per cent sign (%) represents zero or more characters and the underscore (_) corresponds to exactly one character. In regular expressions, the asterisk (*) corresponds to zero or more characters. We opted to use the question mark as a symbol of a wildcard, followed by a number to distinguish between different wildcards (e.g., first wildcard represented by a "?1", etc.)

Modifying a wildcard: Users are able to create a wildcard by first selecting the part of the field contents which must be abstracted. Then, users can right-click to open the Wildcard context

menu and select the "Create wildcard" option. The selection will be replaced by a wildcard icon containing the index of the wildcard. This process is displayed in Figure 4.13. After creation, the wildcard can be used in other places in a similar manner - by selecting the other places and choosing the "Use wildcard" option from the context menu. Wildcards can be removed by right-clicking on a wildcard and selecting the "Remove wildcard" option. If a wildcard is used in multiple places, only the selected instance of the wildcard will be removed.

Figure 4.13: Creating a wildcard. A) an entry field "Location" without wildcards. B) Selected part of the field to be substituted by a wildcard and the Wildcard context menu. C) The resulting Wildcard

Implementation

Because regular expressions were used to implement the Structure Matching functionality, the wildcards are based on regular expressions as well. Specifically, we benefit from regular expression capture groups which allow us to substitute the data that needs to be abstracted with any character. Furthermore, the use of capture groups allows us to use a wildcard in several places by referencing the capture group for that wildcard

4.3 Components

Our solution is implemented in agreement with the React best practices [43] and the design principles documented by the React developers [7] by separating the functionality over several React components [8]. In this way, the solution is modular, clear, and easily maintainable. Furthermore, we prioritise keeping consistent with the code implemented in Tracy by developers of the Accelerando team, both in terms of naming convention and coding approach). To make sure a proficient quality of code was reached, we checked our code with ESLint [9], a static code analysis tool for identifying problematic code and Prettier.io[10], a code formatter enforcing consistent code

[7] https://legacy.reactjs.org/docs/design-principles.html
[8] https://react.dev/reference/react/Component
[9] https://eslint.org/
[10] https://prettier.io/

format. These tools were not used in Tracy prior to our development, but are popular development tools and have been adopted by the Tracy team since. To ensure consistency with the rest of the source code in Tracy, our work was code-reviewed by two developers from the Accelerando team. Figure 4.14 presents the component diagram of Tracy following the implementation of the Structure Matching and the Wildcard Features. Rectangles with rounded corners represent React components, whereas the ones without represent custom hooks (i.e., Typescript files containing logic). Arrows signify a parent-child relationship between the components (i.e., from parent to child) whereas dotted lines represent the use of a custom hook by a component.

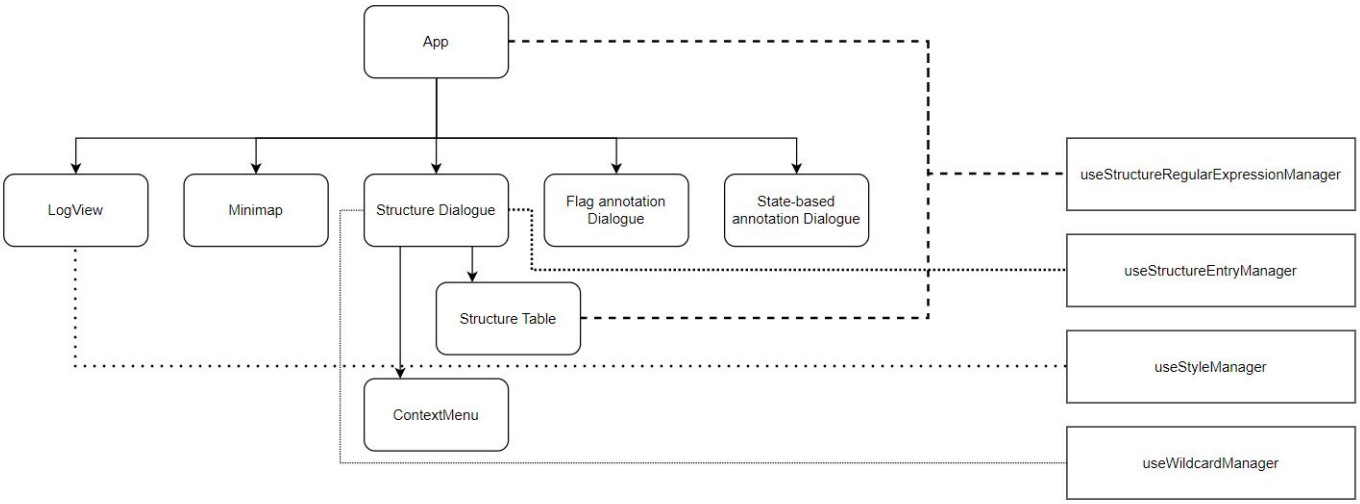

Figure 4.14: Component diagram of Tracy

Chapter 5

Evaluation

To address our final research question (Section 1.2, RQ3), we conducted a user evaluation study. Due to our iterative approach to implementation and evaluation, we evaluated individual features. Moreover, because the character of our study is exploratory, we prioritised exploring the usefulness and usability of our features over other aspects such as performance which can be assessed in future work. The user evaluation study was approved by the Ethical Review Board of TU/e (ref: ERB2023MCS9) and Philips IGTS management.

5.1 Evaluation Setup

5.1.1 Participants

Due to a limited pool of available participants at the time, we opted to recruit participants by using convenience sampling. The participants were selected from the list of employees, provided to us for our interview study (Section 3.1.1). We approached engineers from various teams by sending them an introductory email in which we described that we were evaluating a feature in Tracy, as the platform had been presented in front of many developers from Philips IGTS prior. We approached 14 people in total out of which three participated in the first round of evaluations and two in the second, five were unavailable, three did not respond and one declined to participate. One of the participants of the evaluations also participated in our interviews, while the other four did not. While conducting evaluations with participants who had been previously interviewed could support the validity of our interview findings at the time of the evaluation, only that interviewee was available. Furthermore, we chose to include engineers whom we had not interviewed to avoid confirmation bias [44].

We conducted an individual session with each participant. Each session consisted of four parts. First, we explained the evaluation procedure, also included in the invitation to the session, and we obtained written consent for participation from the participant. Then, we gave a short demo of the evaluated feature, followed by an opportunity for the participant to pose questions. Afterwards, the participant completed a series of tasks using the feature while thinking-aloud [45]. With the completion of these tasks, we wanted to observe the impact of our feature on the process of log analysis. During the completion of tasks, we recorded the screen displaying Tracy and the voice of the participant. We reviewed the recordings after the session to better understand and document the experiences of our participants. After the completion of the tasks, the participant filled in a short questionnaire (Section 5.1.3) regarding the usability of the feature. Finally, we provided the participant with time to give feedback and pose further questions, before

concluding the session. Table 5.1 presents a summary of the evaluation sessions.

Table 5.1: Summary of evaluation sessions

Participant ID	Position	Use of tools for log analysis	Available tool functionality	Type of log used during evaluation
P1	Product owner / Sr. Software Engineer	Excel-based self-made tool	- searching and filtering (full excel functionality)	Functional trace
P2	Software Engineer	self-made tool	- merging several logs, - searching for text, - filtering - showing source code	Functional trace
P3	Software Engineer	self-made tool	- merging several logs, - searching for text, - filtering - showing source code	Functional trace
P4	Software Engineer	UltraEdit	- searching for text in multiple logs	Functional trace
P5	Systems Test Engineer	Excel-based self-made tool	- searching and filtering (full excel functionality)	Event log

* "self-made" refers to tools created by individuals, not tools created by the specific participant

5.1.2 Tasks

With the tasks that participants had to complete, we strived to examine the benefits and disadvantages of using the feature when analysing logs during the investigation of software bugs. Therefore, we asked participants to provide a log file (event log or functional trace) to use during the completion of tasks, in order to evaluate the impact on representative files. We only specified two requirements for the files: that it was a file the participant analysed during a recent investigation and that the file contained information used for locating and fixing the bug.

Each participant was first asked to complete several "warm-up" tasks that involved performing all possible actions of that feature. As the participants had no prior experience with the feature, with these tasks, we aimed to help the participants practice and be able to use the feature during the main tasks competently. The main tasks are based on the two use cases envisioned for the Structure matching feature in Section 4.2.1. Participants in the second round of evaluation completed the warm-up tasks for both structure matching and wildcards. Below we present all the tasks used during the two rounds of evaluations.

- Structure matching tasks (warm-up)
  - Define a structure consisting of one row and search for matching structures.
  - Change the field selection and search again.
  - Navigate through the results.
  - Add an additional row to the structure and search again.

  - Toggle the structure link between the rows and search again.
  - Remove one of the rows from the structure.

- Wildcards tasks (warm-up)
  - Create a wildcard and search for matching structures.
  - Use the wildcard in another field and search for matching structures.
  - Remove the wildcard from one of the fields.

- Main tasks:
  - M1: Try to define a structure that can help you navigate to the entry showing the source of the bug.
  - M2: Try to define a structure that captures entries reoccurring multiple times that are irrelevant to the current investigation.

5.1.3 Questionnaire

To improve the objectivity, reliability, replicability and communication of our evaluation results we used a standardised questionnaire [46]. As the evaluations were conducted iteratively with the implementation work, we opted for a post-task questionnaire as opposed to a post-study one. In particular, we employed the Software Usability Scale (SUS) [47] questionnaire. SUS consists of ten statements for each of which participants must indicate the extent of their (dis)agreement. To express their opinion, participants select an option out of a five-point Likert scale [48]. We adapted the original SUS statements to fit better our context, by substituting the word "system" with "feature". The adapted version of the SUS questionnaire can be found in Appendix A.

All SUS statements indexed with an odd number are written with a positive tone and all the even statements with a negative tone. The change of tone between statements aims to alleviate the risk of acquiescence bias [49]. Responses to all statements are required for scoring the answers. The questionnaire produces a score ranging between zero and a hundred. Lewis et al. [50] benchmark SUS scores obtained from different studies. Based on that benchmark the average score across studies is 68 (average usability) while a score above 80 indicates high usability. Furthermore, there is a significant positive correlation between SUS scores and Net Promoter Score a metric of customer loyalty [51].

5.2 Evaluation

5.2.1 Structure Matching

For the evaluation of the structure matching, we conducted evaluation sessions with three participants from different teams within Philips IGTS. After the third session, we chose to continue with another implementation round based on several observations described below. All three participants were aware of Tracy, its GUI and its other features at the time, because of a demo of the platform, given to engineers from multiple teams at Philips IGTS, several months prior to the evaluation sessions. Nevertheless, two of them (P2, P3) were completely unaware of the structure matching feature and its interface before the sessions and one (P1) knew that such a feature was being developed. To lessen the risk of receiving socially acceptable answers from the participants, known as social desirability bias [52], we did not reveal to participants P2 and P3 that we had developed the feature. Instead, we stated that our goal is to evaluate novel

features of Tracy, developed by the Accelerando team. This approach was not feasible with the first participant due to his prior knowledge.

All participants were able to successfully complete all previously described main tasks. During the completion of task M1, all participants were able to use structure matching to reach entries that displayed the presence of the relevant bug. The origin of the bug was visible in one of the occurrences of the structure defined by P3. The origin of the bug was not present in the files prepared by P1 and P2. Both participants reached an entry showing the presence of a bug in a different component and needed to continue the analysis in logs that they had not prepared for the evaluation. Interestingly, after receiving the first task warm-up task, all participants began defining structures relevant to the specific bug. As we had not explained the concept of a structure or provided any descriptions of the feature apart from the demo given at the beginning of the evaluation we were surprised to see participants defining a structure relevant to their investigations immediately. Additionally, all participants were able to combine the structure matching feature with the text-searching functionality. They did so by searching for text they knew was contained in entries that they wanted to use in a structure. As a result, all participants were able to quickly define structures for completing task M1.

Possible improvements All participants expressed the need to abstract away business and technical data from the fields. This need became apparent during the completion of task M2. All participants were aware of structures which were irrelevant and occurred hundreds of times in the files. These structures contained business and technical data and the participants were unable to match all occurrences of the structures. While the structures defined by P2 and P3 resulted in 100-200 matches, the one defined by P1 produced only 13 matches. In all three cases, the participants expected more matches.

The structures that were defined during the evaluations consisted mainly of two or three entries. P1 and P3 attempted to match a structure consisting of four entries. In the case of P1, that structure consisted of all the steps performed by the process during which the existence of the bug was visible. In the case of P3, the participant defined a structure consisting of two entries. Matching that structure resulted in several occurrences that contained several hundred entries and were difficult to view as the participant had to scroll extensively. Therefore, P3 attempted to add more entries to his structure in an effort to constrict the results of the matching. Nevertheless, the participant was unable to define a structure resulting in smaller occurrences that could be easily navigated. After the evaluation session, P3 stated that in hindsight he could have constricted the structure they defined by changing the used fields. To help new users take full advantage of the feature we could provide access to a help menu containing short descriptions of the actions The use of structures consisting of few entries may be because users were using the feature for the first time. To help new users take full advantage of the feature a help we could provide access to a help menu containing short descriptions of the actions available to users.

SUS scores

The answers on the SUS questionnaire provided by the participants (Figure 5.1) appear to be consistent with our observations of their experience during the completion of tasks. P1 was the most confident and did not experience any difficulties in using the feature, while the other two participants were unable to completely memorise all the available actions of the feature and contemplated how to perform certain steps (e.g., remove an entry from the structure definition, etc.). Participants provided similar answers on all but one statement. While P1 and P2 strongly disagree with statement eight ("I found the functionality very cumbersome to use."), P3 agrees with it. This contradiction may stem from the struggle of P3 to navigate through the results of

the structure matching, which did not occur with other participants.

	Strongly Disagree				Strongly Agree
1. I think that I would like to use this functionality frequently.	1	2	P3	P1, P2	5
2. I found the functionality unnecessarily complex.	P1, P3	P2	3	4	5
3. I thought the functionality is easy to use.	1	2	3	P2, P3	P1
4. I think that I would need the support of a technical person to be able to use this functionality.	P1, P2, P3	2	3	4	5
5. I found the various functions in this functionality were well integrated.	1	2	3	P1, P2, P3	5
6. I thought there was too much inconsistency in this functionality.	P1	P2	P3	4	5
7. I would imagine that most people would learn to use this functionality very quickly.	1	2	3	P2, P3	P1
8. I found the functionality very cumbersome to use.	P1, P2	2	3	P3	5
9. I felt very confident using the functionality.	1	2	P3	P2	P1
10. I needed to learn a lot of things before I could get going with this functionality.	P1, P2	P3	3	4	5

Figure 5.1: Answers of all participants evaluating the structure matching on the questionnaire. The options that are displayed with darker shades of colour were selected more often by the participants than the options that were displayed with lighter shades of colour.

The calculated SUS scores are presented in Table 5.2. The average score (81.7) indicates overall high usability for the structure matching feature.

5.2.2 Wildcard

We contacted ten potential participants for the evaluation of the wildcard feature. However, we held an evaluation session with only two of them as four were away, three did not respond and one declined to participate. Prior to the evaluation sessions, we decided to investigate the impact of the structure matching and wildcards in log analysis performed for a different purpose than investigating software bugs. Therefore, we conducted a session with a Systems Test Engineer (P5). Such engineers do not investigate bugs, however, they analyse event logs containing tests performed during the execution of software.

P4 was able to use the structure matching to reach the source of the bug in the log. Although the structure occurred a single time in the file, the occurrence was large (i.e., many entries were logged between the entries in the structure) and the participant struggled to scroll through it.

Table 5.2: Structure Matching feature - SUS Scores

Participant ID	SUS Score
P1	95
P2	82.5
P3	67.6
Total	**81.7**

Furthermore, as there was only one occurrence of the structure the participant did not need to use wildcards during the structure definition. Nevertheless, the wildcard feature was utilised during the completion of task M2. P4 knew that hundreds of tests were performed and logged in the file which were irrelevant to the investigation. By defining a structure consisting of two entries - one logging the start of such a test and one logging the end, the participant was able to match 417 occurrences. Wildcards were integral to this result as each test-related entry contained several technical values such as the identifier of the test and configuration variables.

P5 did not complete tasks M1 and M2 as they were irrelevant to the analysis performed by Systems Test Engineers. Moreover, the participant used the features on a log he was inexperienced with as the original file he had prepared was corrupted. Therefore, the participant used the features to explore the log and comprehend the contents. The participant was interested in the presence of failed test cases. Hence, they defined a structure consisting of an entry describing the result of a test and used the wildcard to abstract away the test identifier and several other technical data. By doing so they found 3 failing tests in the log. Then, they wanted to inspect the entries logged during the preparation of specific tests. They defined a structure the result of which contained several hundred entries and was difficult to view, similarly as with P3 and P4.

Possible improvements One apparent limitation of our features is that occurrences of structures could contain tens or hundreds of entries. In some cases, users are interested in viewing only the entries that they used when defining the structure and the entries logged in between are of no interest. Therefore, improving the visualisation of the Structure Matching results would improve the impact and the user experience of using the features. Participants have also suggested combining the structure matching feature with other Tracy functionalities such as searching and filtering. One advantage of this combination would be to use structure matching to locate the parts of the log relevant to an investigation and then use the text search and filter to further analyse these parts.

SUS Scores

Participants P4 and P5 chose the same or similar answers when answering the SUS questionnaire on all statements but one (Figure 5.2). While P5 strongly agrees that the feature is easy to use, P4 disagrees. This disagreement may be caused by the difficulty of P4 in comprehending the different constraints on the distances between entries in the structure definition (Section 4.2.1).

Figure 5.2 displays the calculated SUS scores during the evaluation of the Wildcard feature. The average score (75) is lower than that from the previous evaluations (81.7). Nonetheless, it indicates above-average usability.

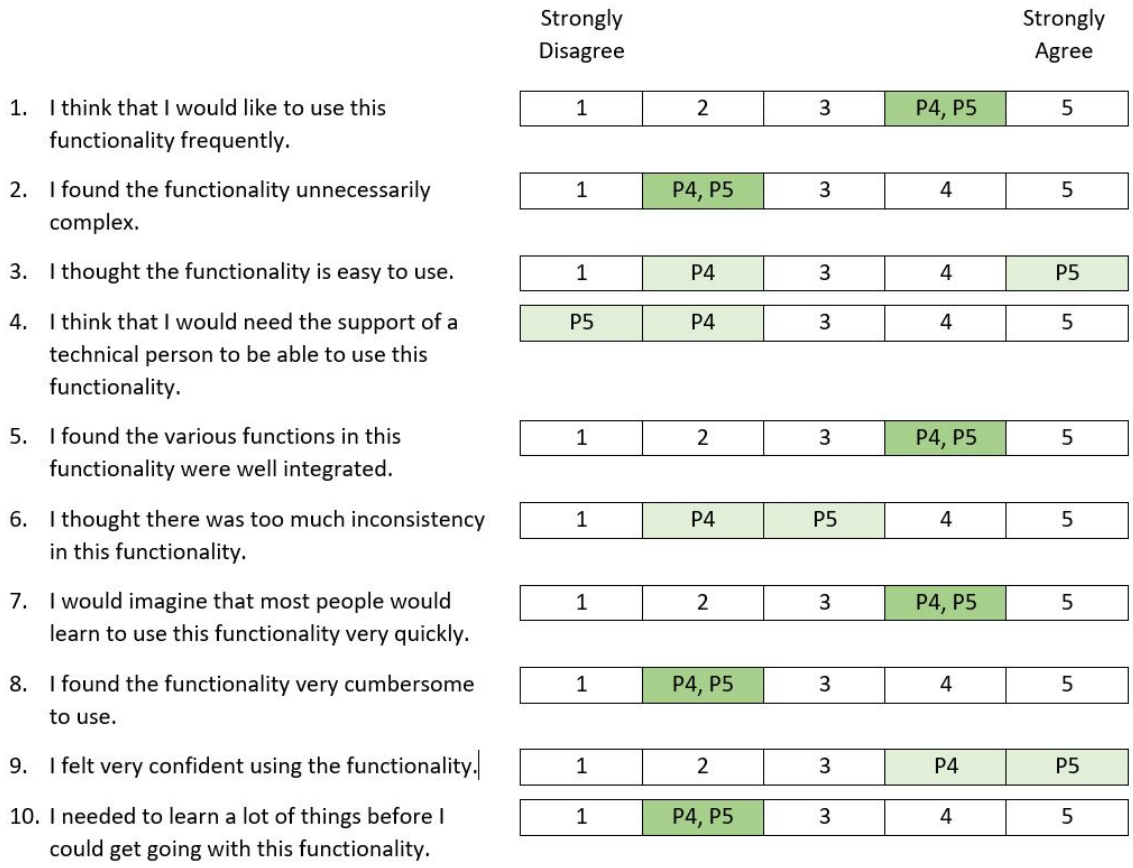

Figure 5.2: Answers of all participants evaluating the wildcard and structure matching on the questionnaire. The options that are displayed with darker shades of colour were selected more often by the participants than the options that were displayed with lighter shades of colour.

Table 5.3: Wildcard feature - SUS Scores

Participant ID	**SUS Score**
P4	70
P5	80
Total	**75**

5.3 Summary of Evaluation Results

Through the evaluations, we have observed that participants were able to utilise the proposed functionality to encode their domain knowledge of structures during the analysis of logs. The Structure Matching and Wildcard features could be used to complete tasks related to both planned use cases - encoding knowledge of structures relevant to an investigation and structures containing unimportant entries or "noise". Furthermore, participants displayed enthusiasm towards the features and indicated an interest in using them in their current practice. These emotions were reflected in the resulting scores of the SUS questionnaire, which indicate above-average usability.

The impact of the features, however, is limited by the knowledge of the users. Therefore, the features may prove most useful on logs produced by software with which engineers are experienced. We shortly explored the use of the features in the context of an analysis performed for a purpose other than bug investigation (i.e., analysing system test cases) and we observed that the features could be used in a different context. These observations, however, must be further investigated and confirmed and are an interesting topic for future work.

Finally, we identified a significant limitation of the implemented features in the visualisation of the results. Specifically, matched structures can contain large numbers of log entries which hinders the value of such results with no way to easily navigate or through them. Addressing such visualisation challenges can provide another topic for future work.

Chapter 6

Threats to validity

In the following chapter, we discuss the threats that could impact the validity of the work presented in this thesis as well as the attempts to mitigate them. We address three categories of threats - construct, internal and external, in accordance with the categories described by Perry et al. [53].

6.1 Construct validity

Threats to construct validity concern the relation between the theoretical concepts (i.e., constructs) under study and the final observations. One such threat is that participants might hold certain associations of the concept of structure, as investigated in this work, to certain patterns or sequences. To mitigate this risk we masked the concept of a structure by excluding questions about it during our exploratory interviews. Instead, we posed questions about the workflow steps taken during log analysis as well as the used information, to observe what structures are inherently present and utilised by participants.

6.2 Internal validity

Threats to internal validity concern the choices made throughout this work and the effects on the outcome. One possible risk is that participants in our interview study might have misunderstood the questions. To mitigate this threat we conducted a pilot interview and modified the selection of questions used for the following interviews accordingly. Additionally, participants could hesitate to share current log analysis challenges or difficulties with the tooling due to various reasons such as fear of punishment. We attempted to reduce this risk by guaranteeing absolute anonymity to our participants and explaining data privacy rights. Furthermore, as the data from the interviews was analysed by a single researcher there is a risk of misinterpretation. To alleviate this threat, we compared our results to those produced by an AI model for analysing interview data.

To reduce the risk of implementation errors, we followed best practices both for the chosen programming language and framework. We utilised widely-used software libraries to improve the robustness of our implementation. Moreover, our implementation was code reviewed by two software developers working on Tracy.

Lastly, participants who evaluated our functionality might suffer from the social desirability bias [52]. For example, they could provide socially acceptable answers. To reduce this risk, we avoided explaining to participants that we had developed the evaluated functionality. Instead,

we explained that our goal is to evaluate functionality developed by the Accelerando team. Furthermore, we employed a standardised questionnaire, used widely in user experience research to avoid the acquiescence bias. Another possible threat is reactivity [54]. For example, participants might complete the tasks during evaluation in an unusual way because they are asked to vocalise their thoughts and are being recorded. To prevent reactivity, we explained that the goal of the evaluations is to evaluate the usability of the software, that the performance of the participant is not relevant and similarly to the interview study we guaranteed full anonymity. Finally, to observe the use of the implemented features on logs used by practitioners, participants utilised our features on logs which they had previously analysed, therefore, there is a risk that the features could not produce similar results on logs with which participants are inexperienced.

6.3 External validity

Threats to external validity are related to the generalisability of the study findings beyond the scope of this thesis. First, for our exploratory interviews and our evaluations, we employed convenience sampling when recruiting participants. We expect that the knowledge, experience and challenges of our participants are representative, as Philips IGTS is representative of a large-scale software development company where complex software systems and services are developed. Nevertheless, due to the concentrated scope of our investigation and the small number of participants we cannot generalise our findings. Further confirmation studies need to be conducted in different environments to corroborate our findings.

An additional threat concerns the contents and format of event logs and functional traces. While different formats of logs are used in Philips IGTS, all logs analysed in Tracy are converted to the Tracy-supported format of JSON[1]. Furthermore, Tracy imposes several prerequisites on logs, all logged entries should contain the same fields and the first field should be a timestamp. These prerequisites guarantee a certain quality of the files. However, in different environments, the log contents may be significantly different. Hence, there is no guarantee that knowledge about inherent structures could be exploited in the same manner as in this thesis.

[1] https://www.json.org/json-en.html

Chapter 7

Conclusions & Future work

7.1 Conclusions

This thesis explores how to encode and use domain knowledge of logs in a visual and interactive way to support log analysis. We focus on the inherent structures in logs and how engineers can leverage their knowledge of these structures during analysis. Previous works propose various tools and techniques for using patterns and visualising logs, but they are rarely used in the software industry. Engineers mainly rely on unspecialized tools, such as text editors and scripts. To understand why and to learn what structures are present in software logs and how are they used, we conducted an interview study with eight engineers from Philips IGTS. We found that they use the concept of structures in their log analysis, however available tools do not support the active use of this knowledge well.

We identified two types of structures that experienced engineers recognise: a high-level structure of a log and a sequence of entries related to a specific process referred to as a sequence or flow. The knowledge of the high-level structure helps engineers narrow down the scope of a bug investigation. The knowledge of the sequences or flows helps them filter out irrelevant entries, form mental models of the software behaviour, and compare logs. Nevertheless, the utilisation of such knowledge was limited due to the varying availability and use of tools.

Based on these findings, we implemented two features that enable engineers to define structures and find their occurrences in logs. We emphasised the ease of use and interactivity of the features to encourage their adoption. We evaluated each feature with several participants to assess its impact and usability. We asked them to complete two tasks based on real-world workflows and challenges. We also used a standardised usability questionnaire to measure their satisfaction with the features.

We found that the participants were able to encode their knowledge of structures successfully and use our features to facilitate the investigation of software issues. However, we also discovered some limitations of the features. First, the usefulness of the features depends on the experience and knowledge of the users. The features may not be helpful for analysing unfamiliar logs. Second, the features may not handle well structures that contain a large number of entries. Specifically, the results may be hard to navigate and use. Despite these limitations, our solution received above-average ratings for usability from the evaluation participants.

7.2 Future work

At the time of writing the functionality described in this thesis is fully integrated with Tracy and so, it is publicly available on the repository of the platform[1]. A member of the Accelerando team demonstrated Tracy to several engineers from ASML, a reputable company that produces lithography machines. The ASML engineers expressed interest in the Structure Matching and other Tracy features. Therefore, one interesting direction for future work is to explore the use of structures and our tools in different companies. Other directions for future work can address the limitations and challenges of this thesis, such as:

- **Improving the visualisation and interactivity of structures:** Future work can tackle the challenge of displaying and navigating structures that contain many entries.
- **Enhancing the scalability of the solution:** Future work can evaluate and improve the performance of our features for large and complex logs.
- **Enabling the sharing of domain knowledge:** While we identified the possibility of supporting the use of structure knowledge by allowing users to share their knowledge, future work can investigate this functionality and its impact on log analysis.
- **Extending the possible structures:** While our features allow users to specify structures that include entries, they do not allow users to specify structures that exclude entries. Therefore, Future work can expand the range of possible structures.

[1] https://github.com/TNO/vscode-tracy

Bibliography

[1] S. D. Suh and I. Neamtiu, "Studying software evolution for taming software complexity," in *2010 21st Australian Software Engineering Conference*, pp. 3–12, 2010. 1

[2] T. Mens, "On the complexity of software systems," *Computer*, vol. 45, no. 8, pp. 79–81, 2012. 1

[3] R. Zeng, Y. Niu, Y. Zhao, and H. Peng, "Software architecture evolution and technology research," in *Advanced Hybrid Information Processing* (S. Liu and X. Ma, eds.), (Cham), pp. 708–720, Springer International Publishing, 2022. 1

[4] D. El-Masri, F. Petrillo, Y.-G. Guéhéneuc, A. Hamou-Lhadj, and A. Bouziane, "A systematic literature review on automated log abstraction techniques," *Information and Software Technology*, vol. 122, p. 106276, 2020. 1, 2

[5] N. Yang, P. Cuijpers, D. Hendriks, R. Schiffelers, J. Lukkien, and A. Serebrenik, "An interview study about the use of logs in embedded software engineering," *Empirical Software Engineering*, vol. 28, no. 2, pp. 1–56, 2023. 1, 14, 16

[6] S. Messaoudi, A. Panichella, D. Bianculli, L. Briand, and R. Sasnauskas, "A search-based approach for accurate identification of log message formats," in *Proceedings of the 26th Conference on Program Comprehension*, pp. 167–177, 2018. 1

[7] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE international conference on web services (ICWS)*, pp. 33–40, IEEE, 2017. 1

[8] F. Wininger, N. Ezzati-Jivan, and M. R. Dagenais, "A declarative framework for stateful analysis of execution traces," *Software Quality Journal*, vol. 25, pp. 201–229, 2017. 1, 7, 8

[9] B. C. Cappers and J. J. van Wijk, "Exploring multivariate event sequences using rules, aggregations, and selections," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 532–541, 2017. 1, 2, 3, 6, 8, 26

[10] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, "Tessla: temporal stream-based specification language," in *Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21*, pp. 144–162, Springer, 2018. 1, 6, 8

[11] S. Kauffman, K. Havelund, and R. Joshi, "nfer–a notation and system for inferring event stream abstractions," in *Runtime Verification: 16th International Conference, RV 2016, Madrid, Spain, September 23–30, 2016, Proceedings 7*, pp. 235–250, Springer, 2016. 1, 2, 3, 6, 8

[12] E. Zgraggen, S. M. Drucker, D. Fisher, and R. DeLine, "(s—qu)eries: Visual regular expressions for querying and exploring event sequences," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, (New York, NY, USA), p. 2683–2692, Association for Computing Machinery, 2015. 1, 2, 3, 6, 8, 26

[13] M. Monroe, K. Wongsuphasawat, C. Plaisant, B. Shneiderman, J. Millstein, and S. Gold, "Exploring point and interval event patterns: Display methods and interactive visual query," *University of Maryland Technical Report*, vol. 11, 2012. 1, 3, 6, 8

[14] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Communications of the ACM*, vol. 59, no. 8, pp. 32–37, 2016. 1, 2, 7, 8, 26

[15] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Inferring hierarchical motifs from execution traces," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 776–787, 2018. 2, 3, 7, 8

[16] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in *2014 IEEE international conference on software maintenance and evolution*, pp. 21–30, IEEE, 2014. 2, 14, 26

[17] R. K. Yin, *Case study research: Design and methods*, vol. 5. sage, 2009. 3

[18] B. Chen and Z. M. Jiang, "A survey of software log instrumentation," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–34, 2021. 5

[19] B. Chen and Z. M. Jiang, "Characterizing logging practices in java-based open source software projects–a replication study in apache software foundation," *Empirical Software Engineering*, vol. 22, pp. 330–374, 2017. 5

[20] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 102–112, IEEE, 2012. 5

[21] S. Johnson, "Query-by-example (qbe)," *Database Management Systems. New York, NY: McGraw-Hill Publisher*, 1999. 6, 8

[22] J. A. Fails, A. Karlson, L. Shahamat, and B. Shneiderman, "A visual interface for multivariate temporal data: Finding patterns of events across multiple histories," in *2006 IEEE Symposium On Visual Analytics Science And Technology*, pp. 167–174, 2006. 6, 8

[23] D. F. Jerding, J. T. Stasko, and T. Ball, "Visualizing interactions in program executions," in *Proceedings of the 19th international conference on Software engineering*, pp. 360–370, 1997. 7, 8

[24] R. Jonk, J. Voeten, M. Geilen, R. Theunissen, Y. Blankenstein, T. Basten, and R. Schiffelers, "Inferring timed message sequence charts from execution traces of large-scale component-based software systems," *Eindhoven University of Technology, Department of Electrical Engineering, Electronic Systems Group, Eindhoven, Tech. Rep. ES Reports*, 2019. 7, 8

[25] K. Wongsuphasawat, J. A. Guerra Gómez, C. Plaisant, T. D. Wang, M. Taieb-Maimon, and B. Shneiderman, "Lifeflow: visualizing an overview of event sequences," in *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 1747–1756, 2011. 8

[26] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2018. 8

[27] B. G. Glaser and A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*. New York: Aldine de Gruyter, 1967. 9

[28] S. Hove and B. Anda, "Experiences from conducting semi-structured interviews in empirical software engineering research," in *11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 10 pp.–23, 2005. 9

[29] P. E. Strandberg, "Ethical interviews in software engineering," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–11, IEEE, 2019. 9

[30] P. Sedgwick, "Convenience sampling," *Bmj*, vol. 347, 2013. 10

[31] J. Saldaña, *The Coding Manual for Qualitative Researchers*. sage, 3rd ed., 2021. 10

[32] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on software engineering*, vol. 25, no. 4, pp. 557–572, 1999. 10

[33] K. M. Aldiabat and C.-L. Le Navenec, "Data saturation: The mysterious step in grounded theory methodology," *The qualitative report*, vol. 23, no. 1, pp. 245–261, 2018. 10

[34] S. van Breukelen, A. Barcomb, S. Baltes, and A. Serebrenik, "" still around": Experiences and survival strategies of veteran women software developers," *arXiv preprint arXiv:2302.03723*, 2023. 11

[35] J. A. Holton, "The coding process and its challenges," *The Sage handbook of grounded theory*, vol. 3, pp. 265–289, 2007. 11

[36] L. Birt, S. Scott, D. Cavers, C. Campbell, and F. Walter, "Member checking: a tool to enhance trustworthiness or merely a nod to validation?," *Qualitative health research*, vol. 26, no. 13, pp. 1802–1811, 2016. 11

[37] B. Shneiderman, C. Plaisant, M. S. Cohen, S. Jacobs, N. Elmqvist, and N. Diakopoulos, *Designing the user interface: strategies for effective human-computer interaction*. Pearson, 2016. 21, 28

[38] O. JE, "Fixing figures for colour blindness," *Nature*, vol. 598, 2021. 24, 26

[39] M. H. Remmer, N. Rastogi, M. P. Ranka, and E. J. Ceisler, "Achromatopsia: a review," *Current Opinion in Ophthalmology*, vol. 26, no. 5, pp. 333–340, 2015. 26

[40] C. Stephen, "Kleene. representation of events in nerve nets and finite automata," *Automata studies*, 1956. 26

[41] A. Farrel, "Routing backus-naur form (rbnf): A syntax used to form encoding rules in various routing protocol specifications," tech. rep., 2009. 26

[42] F. Mascarenhas, S. Medeiros, and R. Ierusalimschy, "On the relation between context-free grammars and parsing expression grammars," *Science of Computer Programming*, vol. 89, pp. 235–250, 2014. 26

[43] C. S. Roldan, *React 18 Design Patterns and Best Practices: Design, build, and deploy production-ready web applications with React by leveraging industry-best practices*. Packt Publishing, 4th ed., 2023. 29

[44] R. S. Nickerson, "Confirmation bias: A ubiquitous phenomenon in many guises," *Review of general psychology*, vol. 2, no. 2, pp. 175–220, 1998. 31

[45] C. Lewis, *Using the" thinking-aloud" method in cognitive interface design*. IBM TJ Watson Research Center Yorktown Heights, NY, 1982. 31

[46] J. C. Nunnally, "An overview of psychological measurement," *Clinical diagnosis of mental disorders: A handbook*, pp. 97–146, 1978. 33

[47] J. Brooke, "Sus: A quick and dirty usability scale," *Usability Eval. Ind.*, vol. 189, 11 1995. 33

[48] I. E. Allen and C. A. Seaman, "Likert scales and data analyses," *Quality progress*, vol. 40, no. 7, pp. 64–65, 2007. 33

[49] S. Moss, "Acquiescence bias," *Unter: http://www. psych-it. com. au/Psychlopedia/article. asp*, 2008. 33

[50] J. R. Lewis and J. Sauro, "Item benchmarks for the system usability scale.," *Journal of Usability Studies*, vol. 13, no. 3, 2018. 33

[51] T. Tullis and B. Albert, "Chapter 6 - self-reported metrics," in *Measuring the User Experience (Second Edition)* (T. Tullis and B. Albert, eds.), Interactive Technologies, pp. 121–161, Boston: Morgan Kaufmann, second edition ed., 2013. 33

[52] P. Grimm, "Social desirability bias," *Wiley international encyclopedia of marketing*, 2010. 33, 39

[53] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: a roadmap," in *Proceedings of the conference on The future of Software engineering*, pp. 345–355, 2000. 39

[54] S. N. Haynes and W. F. Horn, "Reactivity in behavioral observation: A review.," *Behavioral assessment*, 1982. 40

Appendix A

Evaluation form

Please check the box that reflects your immediate response to each statement. Don't think too long about each statement. Make sure you respond to every statement. If you don't know how to respond, simply check box "3".

	Strongly Disagree				Strongly Agree
1. I think that I would like to use this functionality frequently.	1	2	3	3	5
2. I found the functionality unnecessarily complex.	1	2	3	4	5
3. I thought the functionality is easy to use.	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this functionality.	1	2	3	4	5
5. I found the various functions in this functionality were well integrated.	1	2	3	4	5
6. I thought there was too much inconsistency in this functionality.	1	2	3	4	5
7. I would imagine that most people would learn to use this functionality very quickly.	1	2	3	4	5
8. I found the functionality very cumbersome to use.	1	2	3	4	5
9. I felt very confident using the functionality.	1	2	3	4	5
10. I needed to learn a lot of things before I could get going with this functionality.	1	2	3	4	5