DISSERTATION

Defence held on 29/06/2021 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

by

SALMA MESSAOUDI
Born on 2nd July 1991 in Feriana (Tunisia)

LEVERAGING EXECUTION LOGS TO SUPPORT MODEL INFERENCE AND SOFTWARE TESTING

To my family

Abstract

Many software engineering activities process the events contained in log files. However, before performing any processing activity, it is necessary to correctly parse the entries in a log file to retrieve the actual events recorded in the log.

In the case of cyber-physical systems, execution logs are highly important because such systems integrate multiple third-party components where their source code is not always available. This limits the visibility of the system behavior to what is collected in the execution logs. The increasing amount of logs produced by cyber-physical systems calls for 1) more advanced techniques for accurate log parsing, 2) scalable model inference that will enabling efficient program comprehension and, 3) cost-effective software testing to ensuring the quality of complex software systems.

In this dissertation, we propose a set of approaches based on system execution logs to automate cyber-physical system modeling and support system regression testing.

The main research contributions in this dissertation are:

1. An automated approach to accurately solve the log message format identification problem using the NSGA-II algorithm.
2. A novel technique for taming the scalability problem of inferring the model of a component-based system from the individual component-level logs, especially when only limited information about the system is available.
3. An automated technique for slicing complex system test cases using the global resources usage information available in test case execution logs.
4. Two log-based test case prioritization approaches. Starting by building a log-based system model from test cases logs, this model is used to prioritize test cases, in one way, based on their model-based coverage scores and, in another way, based on their predicted mutant killing capability that is learned from a regression model that was initially trained using log-based features.
5. The implementation of the proposed techniques in prototype tools.
6. An extensive empirical evaluation of the proposed approaches.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Lionel Briand, for his invaluable support and guidance. His insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. It is an honor to be part of his team.

I would like to express my gratitude to my co-supervisors, Domenico Bianculli and Donghwan Shin, for their encouragement and inspiration. I am thankful for their devotion, help, and advice. Their treasured support was influential in shaping my project. I have learned a lot from them, and in many cases they helped me regain trust in myself. Also, I would like to express my thanks to Annibale Pannichella for his guidance, insightful comments, and support.

I am grateful to the members of my defense committee: Fabrizio Pastore and Leonardo Mariani for dedicating time and effort to review my dissertation.

My gratitude extends to SES Luxembourg for supporting my work and providing the case study that was the subject of my empirical studies. In particular, special thanks and appreciations go to Raimondas Sasnauskas and Gianluigi Morelli for their valuable time and for providing me with technical guidance and assistance.

Special thanks to my family for always believing in me. Their tremendous understanding and encouragement in the past few years were always the source of my strength.

Last but not least, I would like to thank my colleagues and friends at the Software Verification Lab for the wonderful time we spent together.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Writing software log messages is a well-established programming practice that is used to collect run-time information during a program's execution. Developers carry out logging by inserting into the source code of the system statements that specify which messages and which run-time information to print into the entries of log files. Cyber-physical systems, such as satellite telecommunication systems, produce a large number of log files because they integrate multiple physical components. For these component-based systems, log files are often the only available data source for the developers and practitioners to diagnose and troubleshoot system failures. This is due to the complexity of the integrated components and the fact that the source code is not always available especially with third-party components. Logs are used during various software engineering activities such as performance diagnosis [91], process mining [46, 118], invariant inference [11], and fault localization [128].

All software engineering activities carry out some sort of log analysis, which processes the events corresponding to the entries contained in the log files. But before performing any processing activity, it is necessary to correctly parse the unstructured log messages, to retrieve the actual events recorded in the log. The lack of a structured format for log messages introduced the need for identifying the different templates used in the log messages contained in the log to enable automated data extraction and analysis on a large scale. Solving this problem for cyber-physical systems requires tackling several challenging issues. First, in these systems, log message formats are numerous and changing on a frequent basis. Second, these systems can produce around 120–200 million log entries per hour [86]. Therefore, log message formats need to be identified *automatically* and in a *scalable* and *accurate* way.

Behavior models of software system components play a key role in many software engineering tasks, such as program comprehension [26], test case generation [40], and model checking [22]. Model inference techniques have been proposed as a viable solution to extract finite state models from execution logs. However, existing techniques do not scale well when processing large logs, such as system-level logs obtained by combining component-level logs. Additionally, state-of-the-art techniques cannot infer, from component-level logs, a system-level model that should capture both the individual behaviors of the system's components and the interactions among them. Given the importance of system behavior models, it is crucial to address the scalability problem of existing log-based model inference techniques.

Software testing is another interesting yet challenging software engineering activity. Regression testing, in particular, is arguably one of the most important activities in software testing yet extremely challenging especially in the case of component-based systems because it involves (i) integration and system testing of heterogeneous components; (ii) manual analysis of the execution logs by the engineers, to validate the behavior of the various components. Moreover, the cost-effectiveness of regression testing can be largely impaired by complex system test cases that are poorly designed (e.g., test cases containing multiple test scenarios combined into a single test case) and that require a large amount of time and resources to run. Many techniques have been introduced for cost-effective regression testing, such as test case prioritization and test suite selection. However, the existing techniques tend to rely only on the source code, if available, and give less attention to the execution logs of the system where valuable information about the system states can be used. To our knowledge, the usability of test cases execution logs was not properly studied to support regression testing.

The complexity of component-based systems and the increasing amount of produced logs call for more efficient and accurate techniques to allow log usability in software engineering activities. In this dissertation, we propose a set of approaches based on system execution logs to automate cyber-physical system modeling and support system regression testing. More specifically, since logs contain valuable details about interactions taking place between the system components, event sequences and parameters, different software engineering activity will benefit from these details to reduce their cost and improve their effectiveness. The work presented in this dissertation is motivated by a case study at SES, a world-leading company in the aerospace industry.

1.2 Research Contributions

The ultimate goal of this work is *to investigate the usage of system execution logs to support different software engineering tasks in the context of cyber-physical systems where large amount of logs are being generated and require processing. Precisely, we propose a set of approaches to automate system behavior modeling from components logs and support regression testing, starting from an accurate and efficient log parsing.*

To overcome the challenges related to the log message formats identification problem, we first formulated this problem as a multi-objective optimization one. To accurately identify the different log messages templates, there are two objectives to meet: 1) match as many log messages as possible (i.e., achieve high *frequency* in matching log messages) and; 2) correspond to the largest extent possible to a particular type of event (i.e., achieve high *specificity*). However, these two objectives—high frequency and high specificity—are conflicting. To tackle this problem, we introduced MoLFI (Multi-objective Log message Format Identification), a tool implementing a search-based approach based on a multi-objective genetic algorithm and trade-off analysis. MoLFI applies the Non-dominated Sorting Genetic Algorithm II (NSGA-II [30]) on a given log file to search the space of solutions for a Pareto optimal set of message templates.

To address the scalability problem of inferring a system model from individual component-level logs, we propose a log-based model building approach, called SCALER, that follows a divide and conquer strategy: it first infers a model of each component from the corresponding logs using a state-of-the-art model inference technique, and then it "stitches" (i.e., we do a peculiar type of merge) the individual component models into a system-level model by taking into account the dependencies among the components, as reflected in the logs. The rationale behind this idea is that, though existing model inference techniques cannot deal with the size of all combined component logs, they can still be used to infer the models of individual components, since their logs are sufficiently small. SCALER tames the scalability issues of existing techniques by applying them on the smaller scope defined by component-level logs.

In the context of regression testing, the cost-effectiveness and usefulness of re-executing test cases can be largely impaired by complex system test cases that are poorly designed. One way to mitigate this issue is decomposing com-

plex system test cases into smaller, separate test cases—each of them with only one test scenario and with its corresponding assertions—so that the execution time of the decomposed test cases is lower than the original test cases, while the test effectiveness of the original test cases is preserved. To this end, we propose a novel approach, called DS3 (Decomposing System teSt caSe), which automatically decomposes a complex system test case into separate test case slices. DS3 uses test case execution logs, obtained from past regression testing sessions, to identify "hidden" dependencies in the slices generated by static slicing. Since logs include run-time information about the system under test, we can use them to extract access and usage of global resources and refine the slices generated by static slicing.
Still in the context of regression testing, we investigate the usefulness of test cases logs in achieving an effective test case ordering. We propose LoTeCaP, a tool with two different log-based approaches for test case prioritization. The first approach is a model-based approach where we build a model from the collected logs and then computes certain coverage criteria to be used to prioritize the test cases. The second approach uses a machine learning technique where we build a regression model using different log-based features (extracted from the log-based model), then we use this regression model to predict the number of mutants that will likely be killed by each test case.

To summarize, the main contributions of this dissertation are:

1. MoLFI: An automated approach for log message format identification problem using the NSGA-II algorithm.

2. SCALER: A novel technique for taming the scalability problem of inferring the model of a component-based system from the individual component-level logs, especially when only limited information about the system is available.

3. DS3: An automated technique for slicing complex system test cases using the global resources usage information available in test case execution logs.

4. LoTeCaP: A log-based test case prioritization tool that leverages the test cases logs to characterize their fault detection capabilities. It applies two approaches, one based on model-coverage criteria and one based on predicting the number of killed mutants using a machine learning algorithm.

1.3 Dissemination

Our research work has led to the following publications (listed in chronological order based on their publication date):

- [84]: A Search-based Approach for Accurate Identification of Log Message Formats.
- [105]: Scalable Inference of System-level Models from Component Logs.
- [85]: Log-based Slicing for System-level Test Cases.

1.4 Organisation of the Thesis

The rest of this thesis is organized as follows. Chapter 2 introduces some background concepts which are used throughout this thesis. Chapter 3 presents the log messages format identification problem and the proposed approach. Chapter 4 focus on model inference and how system logs can be leveraged to obtain a scalable inference technique. Chapter 5 illustrates the combination of static slicing and logs in supporting system level testing. Chapter 6 explains how test cases logs can be used for test case prioritization. Chapter 7 discusses the related work and, lastly the conclusions and directions for future work are provided in Chapter 8.

Chapter 2

Background

This chapter contains the different background concepts that are used throughout this thesis.

2.1 System Logs

A log is a sequence of log entries; a log entry contains a timestamp (recording the time at which the logged event occurred) and a log message (with run-time information related to the logged event). A log message is a block of free-form text that can be further decomposed [84] into a fixed part called event template, characterizing the event type, and a variable part, which contains tokens filled at run time with the values of the event parameters. For example, given the log entry `20181119:14:26:00 send OK to comp1`, the timestamp is `20181119:14:26:00`, the event template contains the fixed words `send` and `to`, while the tokens `OK` and `comp1` are the values of the event parameters. More formally, let ET be the set of all events that can occur in a system program P and V be the set of all mappings from events parameters to their concrete values, for all events $et \in ET$; a log l is a sequence of log entries $\langle e_1, \ldots, e_n \rangle$, with $e_i = (ts_i, et_i, v_i)$, $ts_i \in \mathbb{N}$, $et_i \in ET$, and $v_i \in V$, for $i = 1, \ldots, n$. For component-based systems, we denote the log of a component A with l_A and we use the notation $e^A_{i,j}$ for the i-th log entry in the j-th execution; we drop the subscript j when it is clear from the context.

When we have a set of test cases $TC = \{tc_1, tc_2, \dots, tc_n\}$, we denote the set of execution logs of each test case in TC with $\mathit{Logs} = \{l_1, l_2, \dots, l_n\}$.

2.2 Multi-Objective Optimization Problems

This section summarizes basic concepts of multi-objective optimization and briefly describes NSGA-II [30].

A multi-objective problem is an optimization problem that involves *multiple* objective functions.

Let S be the space (set) of all feasible solutions and F be a vector-valued objective function $F\colon S \to \mathbb{R}^k$ composed of k real-valued objective functions $F = (f_1, \dots, f_k)$, where $f_i\colon S \to \mathbb{R}$ for $j = 1, \dots, k$; a multi-objective optimization problem is defined as $\max(f_1(x), \dots, f_k(x))$ subject to $X \subseteq S$. In other words, the problem consists in finding a set of feasible solutions that maximize the objective functions in F.

The goodness of a solution in a multi-objective optimization problem is defined in terms of the *dominance relation* and *Pareto optimality.* More precisely, a solution X is said to *dominate* another solution Y, denoted as $X \prec Y$, if and only if for all indices $i \in \{1, \dots, k\}, f_i(X) \geq f_i(Y)$ and $f_j(X) > f_j(Y)$ for at least one index $j \in \{1, \dots, k\}$. A solution X is called *Pareto optimal* if there does not exist another solution in the search space that dominates it. The set of all Pareto optimal solutions of a given problem is called *Pareto front.* The Pareto front can be used to decide which solution to select, according to the preferences of a decision maker.

NSGA-II The Non-dominated Sorting Genetic Algorithm II (NSGA-II) [30] is a well-known and efficient technique to solve multi-objective problems. NSGA-II is a multi-objective genetic algorithm (GA) that provides well-distributed Pareto fronts and good performance when dealing with up to three objectives [30,61]; it has been widely used in software engineering to solve problems involving multiple objectives [124] and with chromosome representations that require complex data structures (as in our case, see Section 3.4.2).

In NSGA-II (and GAs in general), the candidate solutions to a problem are called *chromosomes.* The encoding of a chromosome depends on the type of problem to solve. GAs refine and evolve randomly-generated chromosomes through subsequent iterations (called *generations*), mimicking selection and reproduction mechanisms in nature.

NSGA-II starts with a pool of randomly generated chromosomes (i.e., *population*). In each generation, the algorithm evaluates the goodness of a chromosome in the current population based on the objectives to optimize. Chromosomes dominating other chromosomes are considered as *fitter* solutions and therefore have higher chances to be selected for reproduction (i.e., for generating new chromosomes). NSGA-II selects the best solutions (parents) within the current population by using *binary tournament selection* [30]. Reproduction is performed by combining pairs of parents to form new chromosomes (called *offsprings*) using two operators: *crossover* and *mutation*. The crossover operator generates two offsprings by exchanging some chromosome parts between the two parents. The mutation operator applies small changes to each offspring to get a more diverse solution. Notice that the implementation of mutation and crossover depends on the problem to solve. The new population for the next generation is formed by selecting the fittest individuals among parents and offsprings according to the dominance relation (non-dominated ranking) and *crowding distance* (to promote diversity) [30]. The process of selecting and recombining chromosomes is repeated multiple times, once for each generation. It terminates either when a given amount of generations is reached or when a time-out occurs. The non-dominated solutions contained in the population of the last iteration represent the final Pareto front.

2.3 Guarded Finite State Machines

We represent the models inferred for a system as guarded Finite State Machines (gFSMs). A gFSM is a tuple $m = (S, ET, G, \delta, s_0, F)$, where S is a finite set of states, ET is the set of system events defined above, G is a finite set of guard functions of the form $g\colon V \rightarrow \{0,1\}$, δ is the transition relation $\delta \subseteq S \times ET \times G \times S$, $s_0 \in S$ is the initial state, $F \subseteq S$ is the set of final states. Informally, a gFSM is a finite state machine whose transitions are triggered by the occurrence of an event and are guarded by a function that evaluates the values of the event parameters. More specifically, a gFSM m makes a guarded transition from a state $s \in S$ to a state $s' \in S$ when reading an input log entry $e = (ts, et, v)$, written as $s \xrightarrow{e} s'$, if $(s, et, g, s') \in \delta$ and $g(v) = 1$. We say that m *accepts* a log $l = \langle e_1, \dots, e_n \rangle$ if there exists a sequence of states $\langle \gamma_0, \dots, \gamma_n \rangle$ such that (1) $\gamma_i \in S$ for $i = 0, \dots, n$, (2) $\gamma_0 = s_0$, (3) $\gamma_{i-1} \xrightarrow{e_i} \gamma_i$ for $i = 1, \dots, n$, and (4) $\gamma_n \in F$.

2.4 Static Slicing

Static slicing is a technique, using def-use analysis [126], for isolating a "slice" of a program (i..e, a subset of the original program statements) that affects the computation of the value of one or more variables in a specific statement in the program. More formally, a slice S of a program P is constructed with respect to a slicing criterion (s, V) where s is a statement in P and V is a set of variables in s; a statement in P is removed to form S if it does not affect the computation of V at s.

Chapter 3

Log Messages Format Identification

Many software engineering activities process the events contained in log files. However, before performing any processing activity, it is necessary to parse the entries in a log file, to retrieve the actual events recorded in the log. Each event is denoted by a log message, which is composed of a fixed part—called *(event) template*—that is the same for all occurrences of the same event type, and a variable part, which may vary with each event occurrence. The formats of log messages, in complex and evolving systems, have numerous variations, are typically not entirely known, and change on a frequent basis; therefore, they need to be identified automatically.

In this chapter, we present the MoLFI approach, which recasts the log message identification problem as a multi-objective problem. MoLFI uses an evolutionary approach to solve this problem, by tailoring the NSGA-II algorithm to search the space of solutions for a Pareto optimal set of message templates.

This chapter is organized as follows. Section 3.1 motivates log messages parsing. Section 3.2 illustrates the problem of log message format identification with an example. Section 3.3 illustrate how log message format identification can be recast as a multi-objective optimization problem. Section 3.4 describes how MoLFI tailors NSGA-II to solve the log message identification problem. Section 3.5 reports on the evaluation of MoLFI. Section 3.6 discusses practical implications, alternative solutions, and limitations of our approach. Section 3.7 concludes this chapter.

3.1 Overview and Motivation

Logging is a programming practice that is used for gathering run-time information of a software system. Developers carry out logging by inserting into the source code of an application statements that specify which messages and which run-time information to print into the entries of log files.

Logging is a pervasive activity: recent studies [135,138] show that between 1/30 and 1/58 of the lines of code in large software systems correspond to logging statements. Furthermore, the importance of logging is also recognized by developers: a recent survey reported that 96% of a group of experienced developers from a leading software company "strongly agree/agree that logging statements are important in system development and maintenance" [42]. Indeed, the information contained in log files can be used for a variety of purposes, such as process mining [46,118], anomaly detection [8,41,44], behavioral differencing [44], fault localization [128], invariant inference [12], performance diagnosis [91], and offline trace checking [6].

All these activities carry out some sort of log analysis, which processes the events corresponding to the entries contained in the log files. Before performing any processing activity, it is necessary to parse the log entries, to retrieve the actual events recorded in the log. A log entry typically includes a timestamp (which records the time at which the logged event occurred) and the actual log message (containing run-time information associated with the logged event). An example of log entry is the following:

```
20050605-06.45.36 send RST CORE to addr 0x0000df30
```

The log message part of a log entry (e.g., the block "`send RST CORE to addr 0x0000df30`" in the above example) is a block of free-form text, which poses a challenge to parsing because it does not have a structured format. More specifically, a log message is composed of two parts: 1) a fixed part, also called *(event) template*[1], which is the same for all occurrences of the same event type; 2) a variable part, which may vary with each event occurrence, containing tokens filled at run time with dynamic information. In the above example, the template contains the fixed words "`send`", "`to`", "`addr`", while "`RST`", "`CORE`" and "`0x0000df30`" are variable tokens. A template is represented as "`send *`

[1]Additional names used in the literature for denoting the fixed part of a log message are "line pattern", "log key", and "message signature".

`* to address *`", where the asterisks indicate placeholders for tokens of the variable part.

The lack of a structured format for log messages leads to the definition of the *log message format identification*[2] problem: given a log file, we want to identify the different templates used in the log messages contained in the log in order to enable automated data extraction and analysis on a large scale.

Solving this problem for complex and evolving systems requires to tackle several challenging issues. First, in these systems log message formats are numerous, changing on a frequent basis (e.g., in Google systems hundreds of new logging statements are added every month [130]), and are typically not entirely known by those who need to analyze log files.

Second, these systems can produce around 120–200 million log entries per hour [86]. Therefore, log message formats need to be identified *automatically* and in a *scalable* way.

Such requirements rule out the use of regular expressions for extracting the templates, since it would still require a manual effort, to create and update regular expressions based on the logging statements contained in the source code of the application. Manual creation and update of regular expressions would be a tedious and error-prone task, given the number of logging statements and the fast pace of their updates [130]. Another strategy would be to statically analyze the application source code, locate logging statements, and extract the templates from the print operations. However, the definition of the static analysis would be tedious and require an extensive knowledge of logging techniques, since logging statements can take different forms in different programming languages and logging frameworks. Furthermore, both strategies outlined above would require to access the source code, which is not always possible, especially in the case of complex software systems that rely on 3rd-party components.

To overcome these limitations, some approaches [33, 41, 49, 78] adopt a black-box strategy that relies on a combination of clustering and heuristic rules to group words into templates, based on their similarity and the frequency within log message blocks. However, our experience on real-world logs shows that these approaches yield low accuracy, as demonstrated by the empirical results reported in this chapter. Furthermore, their parameters (e.g., text similarity) need to be fine-tuned for each log to analyze, usually following a trial-and-error process; these requirements make such approaches neither

[2]This problem is often called "log parsing" in the literature; we believe "log message format identification" is a more specific term, since "log parsing" includes also parsing more structured elements like timestamps and log verbosity levels.

scalable nor effective.

Independently from the specific strategy adopted, any technique for extracting message templates from logs ought to meet two objectives: the generated templates should 1) match as many log messages as possible (i.e., achieve high *frequency* in matching log messages); 2) correspond to the largest extent possible to a particular type of event (i.e., achieve high *specificity*). However, these two objectives—high frequency and high specificity—are conflicting. A template achieving high frequency will contain many tokens in the variable part (to match many log messages), but will be too generic (i.e., it will match messages corresponding to different events); on the other hand, a template achieving high specificity will have a few or no tokens in the variable part (to be able to distinguish between different event types), but it will match only few messages.

Given the presence of conflicting objectives and the limitations of existing solutions [33, 41, 49, 78], in this chapter we propose *to recast the log message format identification problem as a multi-objective optimization problem*, where *frequency* and *specificity* are explicitly considered as two competing objectives to optimize simultaneously. Our approach, named MoLFI (Multi-objective Log message Format Identification), leverages an evolutionary approach to solve this problem. MoLFI applies the Non-dominated Sorting Genetic Algorithm II (NSGA-II [30]) on a given log file to search the space of solutions for a Pareto optimal set of message templates. The two main strong points of MoLFI are: 1) it does not require access to the source code of the application producing the log(s) being analyzed, since it is a black-box technique that works only on the log files; 2) different from existing approaches, it does not require any parameter tuning before its execution.

We implemented MoLFI in a prototype tool. We evaluated the accuracy and efficiency of MoLFI on one proprietary and five publicly-available real-world datasets, containing log files with a number of entries ranging from 2K to 300K; we also compared our approach with IPLoM [78] and Drain [49], two state-of-the-art approaches. The results show that MoLFI achieves by far the highest precision and recall, outperforming the other approaches with substantial improvements in both precision (ranging between $+14\,pp$ and $+86\,pp$, with pp=percentage points) and recall (ranging between $+25\,pp$ and $+75\,pp$) on all datasets, while keeping a running time of less than 126 s when analyzing the largest dataset. A higher accuracy in the identification of log message formats usually has practical implications, in terms of effectiveness, in the log analysis tasks that rely on log message format identification. For example, in the context of anomaly detection—the original motivation for existing work [41, 49]—log analysis is effective only when the parsing accuracy is high

```
20050605-06.45.36 INFO generating core 135
20050605-06.45.36 INFO generating core 198
20050605-06.45.36 INFO generating core 199
20050605-06.45.36 FATAL instruction address 0x0000df30
20050605-06.45.36 FATAL instruction address 0x0000f450
20050605-06.45.36 FATAL machine state register 0x00003000
20050605-06.45.36 FATAL wait state enable 0
20050605-06.45.36 FATAL critical input interrupt enable 0
20050605-06.45.36 FATAL external input interrupt enable 0
20050605-06.45.36 FATAL problem state (0=sup,1=usr)
20050605-06.45.36 FATAL floating point instr. enabled 1
20050605-06.45.36 FATAL machine check enable 1
20050605-06.45.37 FATAL rts internal error
20050605-06.45.37 FATAL rts panic - stopping execution
20050605-06.59.14 FATAL data TLB error interrupt
```

Figure 3.1: Excerpt (simplified) of real-world log entries

enough [48].

To summarize, the main contributions of this chapter are: 1) the formulation of the log message format identification problem as a multi-objective optimization problem; 2) the MoLFI approach for the solution of this problem, based on the NSGA-II algorithm; 3) a publicly-available implementation of MoLFI[3]; 4) the empirical evaluation, in terms of accuracy and efficiency, of the implementation of MoLFI and its comparison with two state-of-the-art approaches.

3.2 The Problem of Log Message Format Identification

We illustrate the problem of log message format identification through the example in Figure 3.1, which provides a simplified excerpt of log entries extracted from an open dataset of logs collected from a BlueGene/L supercomputer system at Lawrence Livermore National Labs.

One can see that the log messages of the first three entries in the example log correspond to the same event type. This event type could be matched with the template ⟨`INFO generating core *`⟩, where the variable part contains one token (indicated with the placeholder `*`). Similarly, entries of log messages at

[3]The evaluation artifacts are available from the following links:
- tool `https://github.com/SalmaMessaoudi/MoLFI.git`;
- log files `https://github.com/SalmaMessaoudi/ICPC-2018-Artifacts.git`.

lines 4–5 could be matched with the template ⟨`FATAL instruction address *`⟩.

However, one could define other templates for the log messages considered above. For example, another template that could match the messages at lines 4–5 would be ⟨`FATAL * * *`⟩, with three tokens in the variable part. Notice that this template is more general than the previous one, since it matches two different types of event (one type associated with messages at lines 4–5, and another type associated with the message at line 13). Another possible template would be ⟨`FATAL * address 0x0000df30`⟩, which is too specific because it matches only the log message at line 4 and misses the message at line 5, even if it is of the same event type.

These examples show that two distinct objectives must be met when identifying message templates:

- maximizing the number of log messages matched by each template, i.e., maximizing the *frequency* of message matches;
- maximizing the *specificity* of a template to a particular type of event.

These two goals are conflicting: to maximize frequency, templates should contain many tokens in the variable part (to match many log messages); however, such templates would have a low specificity (i.e., they would be too generic), matching messages corresponding to different events. On the other hand, to maximize specificity, templates should contain only a few or no tokens in the variable part (to be able to distinguish between different event types); however, they would match only few messages.

Any method proposed to solve the log message format identification problem has to deal with the trade-off between these two conflicting goals.

3.3 Log Message Format Identification as A Multi-Objective Optimization Problem

In this section, we illustrate how log message format identification can be recast as a multi-objective optimization problem and present our approach MoLFI for the solution of this problem, based on NSGA-II.

Problem Formulation As discussed in section 3.2, we consider *frequency* and *specificity* as objective functions to optimize simultaneously. The multi-objective optimization formulation of the log message format identification problem entails that we find, from the set S of all feasible solutions, a set of templates $X = \{\tau_1, \ldots, \tau_n\}$, $X \subseteq S$, such that each template $\tau_i \in X$ with $i = 1, \ldots, n$, matches as many log messages as possible (high frequency) and contains as few variable tokens as possible (high specificity). More formally, the objective functions are the *frequency*: $\mathit{Freq}(X) = \sum_{i=1}^{n} \frac{|match(\tau_i, M)|}{n \times |M|}$, and the *specificity: Spec*$(X) = \sum_{i=1}^{n} \frac{\mathit{fixed}(\tau_i)}{n \times tok(\tau_i)}$, where n is the number of templates in X, M is a list of log messages, $match(\tau, M)$ denotes the list of log messages in M that match a template τ, $\mathit{fixed}(\tau)$ denotes the number of tokens in the fixed part of τ, $tok(\tau)$ denotes the total number of tokens in τ.

When determining a solution to this problem, there are two important aspects to assess. First, the templates contained in a (Pareto optimal) solution may not match all the log messages in M. For example, the solution $X = \{\langle$`FATAL instruction address 0x0000df30`$\rangle$, $\langle$`INFO generating core 135`$\rangle\}$ is Pareto optimal for the log messages in Figure 3.1, since it has the highest possible specificity ($\mathit{Spec}(X) = 1$). However, the templates in X match only two out of the 15 log messages ($\mathit{Freq}(X) = \frac{2}{15}$). Second, two different templates τ_1 and τ_2 in the same solution X may match the same log messages, i.e., $match(\tau_1, M) \cap match(\tau_2, M) \neq \emptyset$. To avoid this type of solutions, we introduce two additional constraints to the optimization problem to determine the set of *feasible solutions* S. More specifically, a solution $X = \{\tau_1, \ldots, \tau_n\} \subseteq S$ is *feasible* if it satisfies the following constraints:

$$\bigcup_{i=1}^{n} match(\tau_i, M) = M \tag{3.1}$$

$$match(\tau_i, M) \cap match(\tau_j, M) = \emptyset \text{ for all } \tau_i, \tau_j \in X, \tau_i \neq \tau_j \tag{3.2}$$

3.4 MoLFI

To solve the multi-objective optimization formulation of the log message format identification problem, we introduce our approach, named MoLFI, which tailors the standard NSGA-II to our context. In particular, we detail the encoding schema and the genetic operators (i.e., crossover and mutation) we use, the pre- and post-processing procedures we apply, and the procedure we follow to select one solution from the Pareto front.

3.4.1 Pre-processing

Before starting the search process, we first pre-process the log messages to improve the accuracy of the process; we follow the guidelines by [48,49]. We first use regular expressions to identify trivial variable parts within the log messages based on domain knowledge, e.g., numbers, memory and IP addresses. Strings in the log messages matching these regular expressions are replaced with a special variable token `#spec#` that cannot be mutated in the later stages of the search. To reduce the computation cost of the template identification process, we filter out duplicated log messages, reducing the number of messages to consider for generating templates. The messages are then tokenized, using blanks, parentheses and punctuation characters as word-separators. Finally, messages are grouped into buckets, with each bucket containing messages that have the same number of tokens; we denote with M_k the bucket/group containing messages with exactly k tokens.

3.4.2 Encoding Schema

In our context, a solution is a set of templates $X = \{\tau_1, \ldots, \tau_n\}$ where each template τ_i corresponds to a group of pre-processed log messages having the same length and sharing all fixed tokens in τ_i. Therefore, each template τ_i is a list of tokens, where each token can be either variable (denoted by the symbols $*$ or `#spec#`) or fixed (i.e., the tokens identified during the pre-processing step).

Although very intuitive, this encoding schema is not efficient for computing the log messages being matched by each template. Indeed, this procedure requires comparing every template against *all* log messages even if most of them have a number of tokens not compatible with what is prescribed by the template. To speed-up the matching process, we design a two-level encoding schema: a chromosome C is a set of groups $C = \{G_1, \ldots, G_{max}\}$, where each group $G_k = \left\{\tau_1, \ldots, \tau_p\right\}$ is a set of templates having the same number of tokens k. This encoding schema guarantees that the matching procedure is applied only for messages and templates of the same length.

Figure 3.2 shows an example of chromosome for the log messages in Figure 3.1 based on our encoding schema. It has four groups of templates with lengths 4, 5, 6, and 12; it also satisfies the constraints for feasible solutions.

3.4.3 Initial Population

MoLFI uses the algorithm *InitialPopulation* (Algorithm 1) for generating the initial population. The algorithm takes as input a set of pre-processed log messages M, the population size N; it returns a population P. Each chromosome is randomly generated inside the loop at lines 4–16: after initializing the chromosome C (line 4), it is iteratively filled with groups of templates (lines

Groups	**Templates**											
k=**4**	FATAL	rts	internal	error								
	INFO	generating	core	*								
	FATAL	instruction	address	#spec#								
k=**5**	FATAL	machine	state	register	*							
	FATAL	wait	state	enable	*							
	FATAL	machine	check	enable	*							
	FATAL	data	TLB	error	interrupt							
k=**6**	FATAL	*	input	interrupt	enable	*						
	FATAL	floating	point	instr	enabled	*						
	FATAL	rts	panic	-	stopping	execution						
k=**12**	FATAL	problem	state	(	*	=	sup	,	*	=	usr	)

Figure 3.2: An example of chromosome for the log messages in figure 3.1.

5–15), one group of templates G_k for each group of pre-processed log messages $M_k \in M$ with the same length k.

For each group of messages $M_k \in M$, the algorithm creates a corresponding group of templates G_k (line 6). Initially, the group G_k is empty and therefore it does not match any log message. The algorithm keeps track of the unmatched messages in the set *unmatched*, initialized with M_k (line 7). Then, a log message is randomly selected from *unmatched* (line 9) and used to generate a template τ (lines 10–12). Template τ is a copy of the original log message with the exception of one single token (randomly selected at line 11), which is replaced with the variable token "$*$" (line 12). The newly generated template is then added to the group G_k and used to update the set of unmatched log messages (line 14). The loop at lines 8–14 terminates when the templates composing the group G_k match all log messages in M_k (i.e., when the set *unmatched* is empty). Since this condition has to be satisfied for each group of messages $M_k \in M$, the chromosome C is a feasible solution. Therefore, Algorithm 1 guarantees that all chromosomes in the initial population satisfy our constraints.

Algorithm 1 InitialPopulation

```
Input: Set of pre-processed log messages $M$
       Population size $N$
Output: Initial population $P$
 1: $P \leftarrow \emptyset$
 2: while $|P| < N$ do
 3:     $C \leftarrow \emptyset$
 4:     for each group $M_k \in M$ do
 5:         $G_k \leftarrow$ create an empty group for templates with length $k$
 6:         $unmatched \leftarrow M_k$
 7:         while $|unmatched| > 0$ do
 8:             $log\_message \leftarrow$ randomly select one message from $unmatched$
 9:             $\tau \leftarrow copy(log\_message)$
10:             $index \leftarrow$ random integer $\in [1; k]$
11:             $\tau[index] \leftarrow$ “*”
12:             $G_k \leftarrow G_k \bigcup \{\tau\}$
13:             $unmatched \leftarrow unmatched \setminus match(\tau, M_k)$
14:         end while
15:         $C \leftarrow C \bigcup \{G_k\}$
16:     end for
17:     $P \leftarrow P \bigcup \{C\}$
18: end while
19: return $P$
```

3.4.4 Mutation Operators

3.4.4.1 Crossover

We implemented the *uniform crossover*, which is one of the most popular crossover operators [108, 111]. It generates two offsprings by shuffling the different characteristics (groups of templates in our case) of the parents. Let $A = \{A_1, \ldots, A_{max}\}$ and $B = \{B_1, \ldots, B_{max}\}$ be the two selected parents where each pair of groups $A_l \in A$ and $B_l \in B$ matches the same pre-processed log messages $M_k \in M$ with length k. The uniform crossover first generates a random binary vector β (called the crossover mask) with a length equal to the number of groups in A and B. Then, the two offsprings O_1 and O_2 are obtained as follows: when the binary element in β for the group with length k is zero, offspring O_1 inherits group A_k while O_2 inherits group B_k; otherwise, O_1 inherits group B_k while O_2 inherits group A_k.

Notice that this crossover operator swaps groups of templates between the two parents without changing the set of templates composing each group. Therefore, it generates offsprings that are feasible solutions: each group $A_k \in A$ and $B_k \in B$ covers all pre-processed log messages $M_k \in M$ and they do not contain overlapping templates (i.e., templates that match the same log messages).

Since A_k and B_k are not modified by our crossover, the properties above are preserved independently from which offspring inherits the two groups.

3.4.4.2 Mutation

After crossover, offsprings are mutated using the mutation operator to randomly change the generated templates. Given a chromosome to mutate $C = \{G_1, \ldots, G_{max}\}$, each group $G_k = \left\{\tau_1, \ldots, \tau_p\right\}$ is mutated with probability $\frac{1}{max}$. A group G_k is mutated by changing one of its templates; the template is mutated by adding or removing variable tokens. In particular, let $\tau = [token_1, \ldots, token_n]$ be the template to mutate; each token is mutated with probability $\frac{1}{n}$. The token $token_i$ is mutated as follows: if it is a fixed one, it is replaced by the variable token "*"; if it is a variable token, it is replaced by a fixed token, which is randomly selected among all fixed tokens in position i of the log messages that match τ; if it is the special token `#spec#` added during the pre-processing, it is not mutated. Therefore, our mutation operator either increases or reduces the number of variable tokens in τ. In the former scenario, it likely increases the frequency of the original template τ; in the latter case it increases its specificity.

Different from the crossover, the mutation operator changes the templates within the chromosome's groups. Therefore, it does not guarantee that the mutated chromosomes satisfy the feasible solution constraints. For this reason, we developed a *correction operator* that (i) removes overlapping templates (i.e., two or more templates matching the same pre-processed log messages), and (ii) adds randomly generated templates if a mutated group G_k does not match all messages in M_k. Random templates are added following the same procedure used at lines 7–14 of Algorithm 1. Notice that the *correction operator* is applied after the mutation operator and it is applied only to the mutated chromosome's groups.

3.4.5 Post-processing

At the end of the search, NSGA-II returns a set of feasible solutions that are Pareto optimal, i.e., representing optimal trade-off between frequency and specificity. Due to the random nature of NSGA-II, Pareto optimal solutions may contain log message templates with spurious variable tokens, i.e., variable tokens that have been inserted by mutation across the generations but that do not contribute to match more pre-processed log messages. For this reason, MoLFI post-processes the templates in each Pareto optimal chromosome with a greedy procedure, which iteratively removes all variable tokens

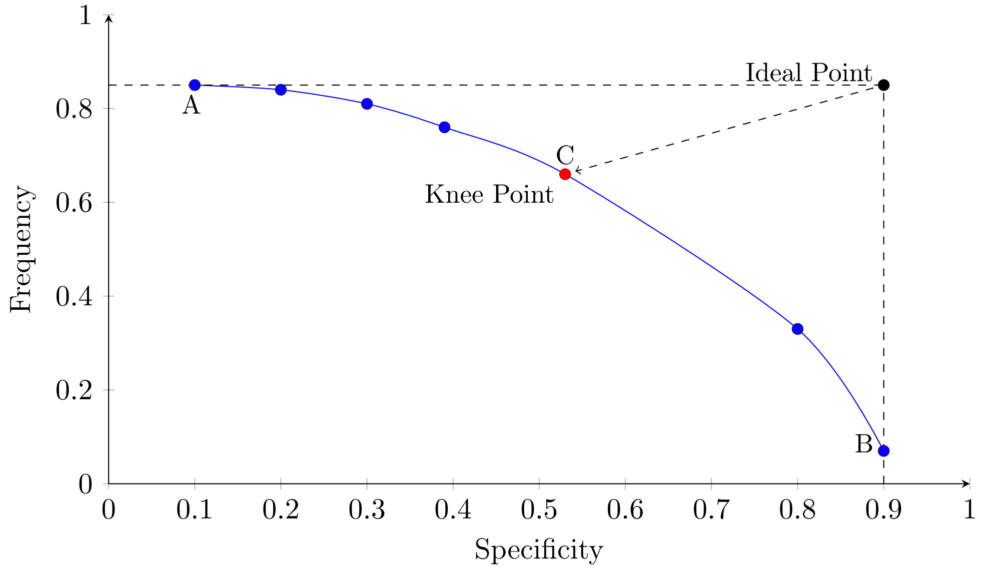

Figure 3.3: The concepts of Pareto front and knee point.

that do not affect the frequency scores. In other words, given a template τ to post-process, the procedure temporarily removes one of its variable tokens and checks whether the set of log messages matched by τ remains unchanged. If the applied change affects the set of matched log messages, the change is reverted; otherwise it is maintained. The post-processing procedure ends once all variable tokens in τ have been verified.

3.4.6 Choosing a Pareto Optimal Solution

If the number of solutions in the generated Pareto front is large it may be difficult to choose one solution (best trade-off) among the different alternatives. For this reason, researchers proposed various guidelines to find and suggest points of interest in the Pareto front, such as the *knee points* [15], *mid points* [92], or the best point (*corner*) for each objective [95].

According to Branke et al. [15], the most interesting Pareto optimal solution is the knee point because any other solution in the front leading to a small improvement in one of the two objectives (e.g., *Freq*) would lead to a large deterioration in the other objective (e.g., *Spec*). To provide a graphical interpretation of the knee point, Figure 3.3 depicts an example of Pareto front for the log message format identification problem. The Pareto front is composed of seven non-dominated solutions: points A and B are the corner solutions of the front while the other solutions represent intermediate trade-offs. Point C can be considered as a knee point since any marginal improvement to *Freq* will correspond to a large deterioration in *Spec*, and vice versa. Therefore, the knee point leads to the lowest loss in both objectives.

To identify the knee point, we measure the distance of each Pareto optimal solution from the *ideal point* [108]. The coordinates of the ideal point correspond to the maximum objective values among all solutions in the Pareto front, considering each objective function separately. For example, for the Pareto front in Figure 3.3, the ideal point has the coordinates (F_{max}, S_{max}), where $F_{max} = 0.85$ (from point A) and $S_{max} = 0.9$ (from point B). More formally, given a Pareto front $P = \{C_1, \ldots, C_p\}$, the knee point $C_{knee} \in P$ is the solution minimizing the distance $\sqrt{(F_{max} - Freq(C_i))^2 + (S_{max} - Spec(C_i))^2}$, for all $C_i \in P$.

3.5 Evaluation

We have implemented the MoLFI approach as a Python program. In this section we report on the evaluation of the effectiveness of the MoLFI implementation in identifying accurate log message formats.

First, we want to assess the performance of MoLFI in comparison with state-of-the-art techniques, in terms of accuracy and efficiency. Second, there are various factors that may influence the effectiveness of MoLFI, such as (1) the number of templates to identify, (2) the population size in NSGA-II; (3) the removal of duplicate messages from the log file to analyze, performed as part of the pre-processing step; we want to understand whether and to what extent these factors affect the effectiveness of MoLFI. Last, in MoLFI we choose the knee point as most valuable solution from the Pareto front, following the general guidelines by [15]. However, different trade-offs in the Pareto front may provide equal or better results in our context; hence, we want to assess whether the knee point is the best Pareto optimal solution for the log message format identification problem.

Summing up, we investigate the following research questions:

RQ1: *How does MoLFI perform when compared to state-of-the-art techniques for the log message format identification problem?*

RQ2: *Which factors impact the effectiveness of MoLFI?*

RQ3: *Is the knee point the best solution to choose from the Pareto front?*

3.5.1 Benchmark

To evaluate MoLFI, we used a benchmark composed of six different datasets: five datasets are publicly available and have been used in previous work on the log message format identification problem [48, 49], while the last one is industrial and proprietary.

The five public datasets are HDFS, BGL, HPC, Zookeeper (shortened to "ZK") and Proxifier (shortened to "PRX"). HDFS consists of logs from the Hadoop file system that were collected from a 203-node cluster on the Amazon EC2 platform [48]. BGL contains logs generated from the Blue Gene/L (BGL) supercomputer, collected by the Lawrence Livermore National Labs (LLNL) [48]. The logs contained in the HPC dataset were collected from a high-performance cluster with 49 nodes and thousands of cores [48]. The logs in ZK were collected by [48, 49] from a 32-node cluster. PRX consists of logs generated by a standalone software [48].

The proprietary dataset (named *PR*) has been provided by one of our industrial partners, active in the aerospace industry; it contains logs produced by a complex system with more than 20 distributed processes.

All datasets contain log files of various size. For the HDFS, BGL, HPC, ZK, and PRX datasets, we used the same samples of 2K log entries used in previous studies [48, 49]. In addition, we also selected a sample of 100K log entries from BGL and a sample of 60K log entries from HDFS. As for the proprietary dataset, we considered three different log files, generated by three different sub-systems, containing 2K, 20K, and 300K log entries.

Ground truth definition. In the case of the log message format identification problem, the ground truth is represented by the actual log message templates. For our evaluation, we established the ground truth as follows.

For the log files with 2K log entries of the public datasets, we used the ground truth defined by [48, 49] and publicly available from their replication package. In the case of the BGL and HPC datasets, the original set of correct templates contains some mistakes, e.g., templates with unbalanced parentheses and missing punctuation marks. Therefore, we manually validated and fixed them before performing our evaluation.

No ground truth is available for the proprietary logs, the 100K log file from BGL, and the 60K log file from HDFS. Therefore, we had to manually establish the ground truth. Two validators independently inspected each log file and extracted the corresponding templates. Then, the two sets of templates independently extracted by the two validators were merged into a single ground truth set, by including only the templates extracted by both validators. Templates identified by only one of the two validators were discussed and further

Table 3.1: Precision (Prec), recall (Rec), F-measure (F-m), and execution time (T(s)) of the three approaches.

Dataset	Size	#T	NTT	Drain				IPLoM				MoLFI			
				Prec	Rec	F-m	T(s)	Prec	Rec	F-m	T(s)	Prec	Rec	F-m	T(s)
BGL	2K	114	31%	0.55	0.51	0.53	0.54	0.47	0.46	0.46	0.28	0.83 ±0.03	0.86 ±0.03	0.84 ±0.03	17.38 ±0.32
	100K	57	22%	0.60	0.74	0.66	32.89	0.42	0.53	0.47	5.76	0.83 ±0.03	0.78 ±0.04	0.80 ±0.03	10.91 ±0.11
HDFS	2K	16	93%	0.86	0.75	0.80	0.29	0.86	0.75	0.80	0.27	1.00 ±0.00	1.00 ±0.00	1.00 ±0.00	3.36 ±0.02
	60K	62	43%	0.79	0.68	0.73	2.11	0.56	0.45	0.50	2.87	0.94 ±0.02	0.95 ±0.01	0.94 ±0.01	10.32 ±0.09
HPC	2K	42	33%	0.49	0.61	0.54	0.32	0.37	0.52	0.43	0.27	0.92 ±0.02	0.88 ±0.04	0.90 ±0.03	7.56 ±0.32
PR	2K	286	14%	0.60	0.64	0.62	0.27	0.52	0.51	0.51	0.40	0.77 ±0.04	0.82 ±0.03	0.79 ±0.03	36.80 ±0.65
	20K	394	17%	0.61	0.57	0.59	1.51	0.54	0.47	0.50	2.38	0.71 ±0.03	0.82 ±0.02	0.76 ±0.02	79.59 ±2.86
	300K	52	80%	–	–	–	–	0.08	0.13	0.10	49.49	0.94 ±0.01	0.88 ±0.00	0.91 ±0.01	125.84 ±4.14
PRX	2K	13	61%	0.46	0.46	0.46	0.21	0.45	0.38	0.42	0.26	0.77 ±0.00	0.77 ±0.00	0.77 ±0.00	3.46 ±0.09
ZK	2K	47	30%	0.77	0.77	0.77	0.22	0.47	0.46	0.46	0.28	0.93 ±0.03	0.87 ±0.01	0.90 ±0.02	6.94 ±0.26

added to the ground truth only upon agreement between the validators. At the end of the validation process, we also verified that no log message in our datasets could be matched by more than one single template in the ground truth. In total, 486K log messages were manually inspected to establish the ground truth. The number of log message templates in each log file ranges from 13 (PRX) to 394 (PR with 20K messages).

3.5.2 Effectiveness of MoLFI

To answer RQ1, we assess the performance of MoLFI, in terms of accuracy and efficiency, in comparison with DRAIN [49] and IPLoM [78], which are the two most recent and effective tools for the log message format identification problem [48, 49]. We use the implementation of DRAIN available in [49] and the one of IPLoM available in [48].

3.5.2.1 Methodology

The NSGA-II algorithm used in MoLFI requires to set four parameters: crossover probability, mutation probability, population size, and stopping condition. To set these parameters, we followed the guidelines proposed in the literature. More specifically, [3] and [103] have empirically demonstrated that the benefits of fine-tuning the parameters of search-based algorithms often do not compensate for the required overhead; both studies recommend to use the default parameters values, since they provide competitive results.

We set the NSGA-II parameters as follows:

- *crossover probability* $p_c = 0.70$, since the recommended values are within the interval $0.45 \le p_c \le 0.95$ [17,23];
- the *mutation probability* p_m is proportional to the length of the chromosome (see section 3.4.4.2), as recommended in the related literature [30];
- the *population size* is set to 20 individuals; according to our preliminary experiments (see section 3.5.3.2), this small value corresponds to the best compromise between accuracy and efficiency;
- the *stopping condition* is set to 200 generations [30].

As selection operator, we used binary tournament selection [30], which is based on dominance and crowding distance.

For DRAIN, in the case of the public datasets, we used the same parameters values used in [49]; in the case of our proprietary dataset, we used the default parameter values: *depth* = 4, *similarity* = 0.5. We also pre-processed the logs, as suggested in [49], to identify trivial variable parts within the log messages based on domain knowledge.

For IPLoM, we used the default parameter values used in [78]: *file support threshold* = 0, *partition support threshold* = 0, *upper bound* = 0.9, *lower bound* = 0.25, and *cluster goodness threshold* = 0.35.

We ran the three tools on each log file in our benchmark and collected the generated log message templates. We measured the accuracy of each tool by comparing the set of generated templates with the ground truth. Furthermore, we measured the wall-clock time for executing the complete program (including pre- and post-processing tasks for MoLFI). To measure the accuracy, we used the metrics used in previous studies [48,49], i.e., $Precision = \frac{|CRT \cap GEN|}{|GEN|}$, $Recall = \frac{|CRT \cap GEN|}{|CRT|}$, and $F\text{-}measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$, where GEN denotes the set of templates generated by a tool and CRT denotes the set of templates generated by a tool which are correct, i.e., conform to the ground truth.

To account for the random nature of NGSA-II, we executed MoLFI 50 times on each log and computed the median and standard deviation of the effectiveness metrics; DRAIN and IPLoM were executed only once due to their deterministic nature. However, DRAIN generated duplicated templates, i.e., multiple templates having exactly the same fixed and variable tokens. To avoid any bias due to duplicated templates, we detected and removed them before computing the various effectiveness metrics.

Furthermore, we used the Welch's t-test to verify whether the F-measure scores achieved by MoLFI are significantly higher than those achieved by the alternative tools. The Welch's t-test is a test for statistical significance suitable for distributions with different variance. In our case, the variance for DRAIN and IPLoM is zero as they are deterministic; MoLFI may return a non-zero variance due to NSGA-II. For this test, we consider a level of significance α=0.05. Other than simply testing the statistical significance, we estimated the magnitude of the differences (effect size) using the Vargha-Delaney ($\hat{A}_{12}$) statistic [120]. $\hat{A}_{12}$ takes values in $[0; 1]$; $\hat{A}_{12} > 0.50$ values indicate that MoLFI outperforms the alternative tool while for $\hat{A}_{12} < 0.50$ the contrary is true. $\hat{A}_{12} = 0.50$ if the two tools are equivalent.

3.5.2.2 Results

Table 3.1 shows the results of the three tools, grouped by dataset and log file size. Column "#T" indicates the number of templates in a file; column "NTT" indicates the percentage of templates with more than one variable token in a file; columns "Prec", "Rec", "F-m", "T" indicate, respectively, the precision, recall, F-measure, and the execution time in seconds. For MoLFI, the table reports the median results achieved across the 50 runs as well as the corresponding standard deviation values.

According to our results, MoLFI obtains, on all the log files in the benchmark, a better F-measure than both DRAIN and IPLoM.

We compared the effectiveness of our approach with the two state-of-the-art tools and we present the differences in percentage points (pp). The difference between MoLFI and DRAIN in terms of F-measure ranges between $+13\,pp$ and $+36\,pp$. The values for the difference are always statistically significant according to the Welch's t-test (all p-values are lower than 0.01) and the effect size is always *large* (i.e., $\hat{A}_{12} \approx 1$). This difference is due both to better precision and to better recall. We also remark that DRAIN crashed on the 300K log from the proprietary dataset, without yielding any message template.

The difference between MoLFI and IPLoM in terms of F-measure ranges between $+20\,pp$ and $+81\,pp$. For all logs in our study, the differences are statistically significant (p-values are always lower than 0.01) with a *large* effect size ($\hat{A}_{12} \approx 1$). Also in this case, the better F-measure is ascribable to the substantial improvements in both precision (ranging between $+14\,pp$ and $+86\,pp$) and recall (ranging between $+25\,pp$ and $+75\,pp$). An interesting case is represented by the 300K log from the proprietary dataset: MoLFI generates very accurate templates achieving an F-measure of 0.91 while IPLoM obtains a very low F-measure of 0.1.

Figure 3.4 shows an example of the Pareto front generated by MoLFI for the dataset HPC on one single run. It also displays the knee point (red point) and two further points (in black color) corresponding to the frequency and

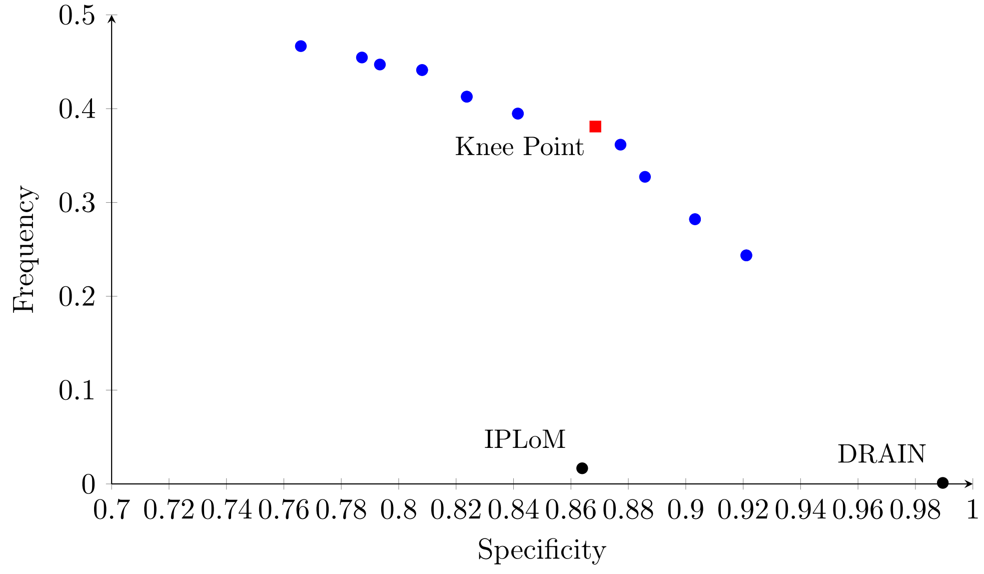

Figure 3.4: Pareto front generated by MoLFI and the objectives scores of the templates generated by DRAIN and IPLoM

specificity values of the templates generated by DRAIN and IPLoM. The knee point dominates the templates produced by IPloM, meaning that MoLFI generates templates having both better frequency and better specificity. Instead, the templates produced by DRAIN are non-dominated neither by the knee point nor by the other Pareto optimal solutions. Indeed, their objective scores are located in one of the corners of the Pareto front, meaning that their specificity is very high (few variable tokens) but their frequency is very low. Similar results are obtained also for the other logs in the benchmark. To sum up, both IPLoM and DRAIN are not able to provide optimal compromises between the two objective functions.

In terms of efficiency, MoLFI is the slowest technique; this can be explained because of the usage of NSGA-II, which is an iterative algorithm. The fastest technique is IPLoM, which, however, is also the one with the lowest F-measure values. DRAIN is faster than MoLFI in all the cases with the only exception of the 100K log from the BGL dataset: for this file DRAIN takes 32.89 s while MoLFI takes only 10.91 s. Although MoLFI takes longer to converge than state-of-the-art tools, the increment of the running time has no practical implications since it took less than 126 s when analyzing the largest dataset.

3.5.3 Factors Influencing the Effectiveness of MoLFI

To answer RQ2, we investigate the effect of the following three factors on the effectiveness of MoLFI: (1) the number of templates to identify in a log file, (2) the population size in NSGA-II; (3) the removal of duplicate messages from the log file to analyze, performed as part of the pre-processing step. In the following, we illustrate the evaluation methodology and the results for each of these factors.

3.5.3.1 Number of Templates

To test the effect of this factor, we used the one-way permutation test [5] to assess whether the number of templates to identify in a log file statistically interacts with the F-measure scores achieved by MoLFI. The permutation test is a non-parametric test and therefore it does not assume that the data are normally distributed. We ran this test with a very large number of iterations (i.e., 10^8), as suggested in the literature [5].

According to the one-way permutation test, there is no interaction between the number of templates to be identified in a log file and the F-measure values obtained by MoLFI (p-value=0.08). This means that our technique yields high F-measure scores both with log files containing a low number of templates (e.g., see the 2K log from PRX in Table 3.1) and with log files containing a high number of templates (e.g., see the 20K log from PR in Table 3.1).

3.5.3.2 Population Size

Given the nature of NSGA-II, using a large population size may significantly increase the execution time for finding the best solutions; however, using a population with few individuals may yield poor results. We test the effect of this factor by running MoLFI (on each log file of the benchmark) with a population size of 40 and 80 individuals. We repeated each run ten times and computed the median F-measure and execution time. We compared these results with those obtained by the baseline (with a population of 20 individuals, see Table 3.1).

Table 3.2 shows the results of this comparison. Column "T" indicates the median execution time in seconds; column "R" is the ratio between the execution time achieved by the new configurations and the baseline; column "F-m" is the median F-measure; column "$\Delta_{\text{F-m}}$" indicates the difference, in percentage points, between the F-measure achieved by the new configurations and the baseline. All these values are shown for the columns labeled "pop=40" and "pop=80" of Table 3.2.

In terms of F-measure, MoLFI performs almost equivalently for the three configurations, with an increase for the majority of log files reaching 4 *pp* for the 2K log file from BGL. We remark two exceptions where the F-measure decreased with a population size of 80: the 300K log file from PR (-5 *pp*) and the BGL log file of size 100K (-1 *pp*).

Execution time sharply increases as the population size grows. This is to be expected since a larger population size entails more fitness computations for NSGA-II.

3.5.3.3 Removal of Duplicate Messages

In the pre-processing step presented in section 4.3.1, we filter out duplicated log messages, to reduce the number of log messages to consider for generating templates. However, such a reduction may also directly affect the value of one of our two objective functions, *frequency*, which could be further reflected in changes to the shape of the Pareto front and to its knee point.

To test the effect of this factor, we ran MoLFI by disabling the routine responsible for removing duplicated log messages in the pre-processing step. As above, each run was repeated ten times and we computed the median F-measure and execution time, as well as the ratio between execution times and the difference in percentage points of the F-measure. The results are shown in the column "no filtering" of Table 3.2. No data are reported for the 300K log from the PR dataset, since it timed-out (> 3 hours) when completing the first generation of NSGA-II.

We compare these effectiveness scores with those reported in Table 3.1 (i.e., with filtering enabled). We observe that the F-measure scores obtained by the two configurations are the same for all log files, with only 1 *pp* increase for the BGL dataset. These results show that filtering out duplicated log messages during pre-processing does not significantly alter the final F-measure. However, it results in a substantial reduction of the execution time. For example, when the filter is enabled, MoLFI requires 126 s to converge for the largest log file (the 300K log file from the PR dataset), while it times out (after three hours) for the same log file when the filter is disabled.

3.5.4 Is the Knee Point the Best Solution?

To answer RQ3, we analyze, over the entire benchmark, the F-measure scores achieved by all solutions in the Pareto front. This means comparing the F-measure of the knee point with the scores achieved by the other Pareto optimal solutions. For the sake of analysis, for each log, we selected one single Pareto

Table 3.2: Comparison between three different configurations of MoLFI: with population size of 40 (column "pop=40") and 80 (column "pop=80") individuals, and without filtering the duplicated log messages (column "no filtering"). "T": median execution time in seconds, "R": ratio of execution time values, "F-m": median F-measure, $\Delta_{\text{F-m}}$: difference of F-measure in percentage points

Dataset	**Size**	pop=40				pop=80				no filtering			
		T (s)	**R**	**F-m**	$\Delta_{\textbf{F-m}}$ (*pp*)	**T** (s)	**R**	**F-m**	$\Delta_{\textbf{F-m}}$ (*pp*)	**T** (s)	**R**	**F-m**	$\Delta_{\textbf{F-m}}$ (*pp*)
BGL	2K	35.84	2.06	0.86	2	70.97	4.08	0.88	4	25.80	1.48	0.85	1
	100K	17.88	1.64	0.80	0	31.90	2.92	0.79	−1	397.76	36.46	0.81	1
HDFS	2K	6.36	1.89	1.00	0	12.48	3.71	1.00	0	13.21	3.93	1.00	0
	60K	18.26	1.77	0.94	0	34.32	3.33	0.94	0	175.09	16.97	0.94	0
HPC	2K	15.33	2.03	0.90	0	29.75	3.94	0.92	2	15.43	2.04	0.90	0
PR	2K	75.44	2.05	0.80	1	162.37	4.41	0.79	0	45.39	1.23	0.79	0
	20K	157.88	1.98	0.78	2	315.11	3.96	0.76	0	217.43	2.73	0.76	0
	300K	155.58	1.24	0.91	0	235.69	1.87	0.86	−5	>3h	–	–	–
PRX	2K	6.55	1.89	0.77	0	12.88	3.72	0.77	0	14.47	4.18	0.77	0
ZK	2K	13.86	2.00	0.90	0	26.56	3.83	0.90	0	15.05	2.17	0.90	0

front among those obtained with 50 independent runs. For the selection, we first computed the F-measure for the knee point generated in each run; then, we selected the knee point having the median F-measure across the runs and its corresponding Pareto front.

Figure 3.5 shows, for all the log files in our benchmark, the F-measure of the knee points (indicated with red points) when compared with all the Pareto optimal solutions (represented by the boxplots). One can see that, in all cases, the F-measure score of the knee point is located at the very top of the boxplot. This confirms our conjecture that, in the context of the log message format identification problem, the knee point is the best solution to choose from the Pareto front.

3.6 Discussion

Practical implications

As discussed in Section 5.4.2, MoLFI achieves a substantial higher accuracy than alternative algorithms. Such improvements in accuracy represent a considerable reduction in the time needed by analysts to inspect the generated

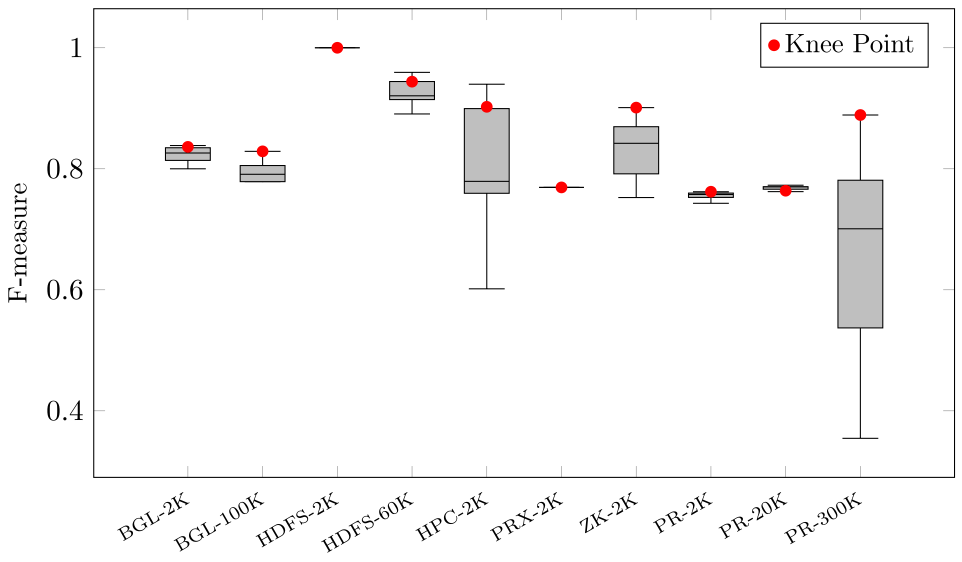

Figure 3.5: Comparisons between the knee-point and other Pareto front solutions in terms of F-measure

templates, validate them and eventually modify the incorrect ones. For example, MoLFI generates 62 templates for HDFS with the 60K log; on average across runs, 60 templates are correct (as they match the ground truth). One incorrect template is T=⟨`PacketResponder * for block * *`⟩, which is too general as it matches log messages belonging to two different log events: (i) when the `PacketResponder` for a given block terminated correctly and (ii) when it has been interrupted. The log messages for these two log events are very similar as they differ only by one single token. Fixing this template would need to create two templates, each one with an additional fixed token. Since T matches only those two log events, fixing it is trivial.

Sum-scalarization vs. multi-objective search

An alternative search-based solution to solve our multi-objective problem would be applying *sum scalarization* [29]. Such a strategy combines the objectives to optimize into one single function by using the sum operator and thus enabling the use of a single-objective genetic algorithm to optimize the aggregated function. In our case, the aggregation function combines frequency and specificity, i.e., $f(X) = Freq(X) + Spec(X)$. To assess this alternative search strategy, we ran a classical genetic algorithm to optimize the function $f(X)$ mentioned above on HDFS with the 60K log. The median F-measure obtained over 10 independent runs is 0.65 (± 0.02), which is statistically significantly lower than the value achieved by MoLFI (i.e., NSGA-II and the knee point), which is 0.94. Note that for the single-objective genetic algorithm we use the same parameter values as for NSGA-II.

The role of constraints

In our problem formulation, we consider two constraints: (i) each log message has to be matched by only one template in a solution *X*; and (ii) the templates in *X* have to match all log messages (100% coverage). The former constraint is straightforward since templates in a given solution *X* should not overlap; the latter is less intuitive but analysts, for some specific applications, could be interested in solutions not covering all log messages. However, we observed that the solutions obtained when removing the coverage constraint have only one single template. For example, if we run MoLFI on HDFS with 60K logs by disabling the coverage constraint, NSGA-II returns a knee point which is a solution with only one template having one variable token and 12 fixed tokens. Such a template has high frequency (*Freq*=0.06) and high specificity (*Spec*=12/13=0.94). No other template is included in the solution because adding any other template would penalize both frequency and specificity.

Limitations

Our approach may produce incorrect results because of the method we use to group messages. In particular, log messages whose variable part has a variable composition (e.g., because of a variable-length argument list), could lead to different templates even if they have the same fixed part.

3.7 Conclusion

The *log message format identification problem* deals with the identification of the different templates used in the log messages. In this chapter, we formulated this problem as a multi-objective optimization one, where the goal is to generate log message templates with high frequency (i.e., they match as many log entries as possible) and high specificity (i.e., specific for each log event). To tackle the problem, we introduced MoLFI, a tool implementing a search-based approach based on a multi-objective genetic algorithm and trade-off analysis.

An empirical study involving six real-world datasets (five publicly-available and one proprietary) showed that MoLFI (i) achieved significantly higher accuracy than DRAIN and IPLoM, two state-of-the-art tools; (ii) is highly scalable to large logs since it requires slightly above two minutes to analyze hundreds of thousands of messages.

Chapter 4

Scalable Inference of System-level Models from Component Logs

Behavioral software models play a key role in many software engineering tasks; unfortunately, these models either are not available during software development or, if available, they quickly become outdated as the implementations evolve. Model inference techniques have been proposed as a viable solution to extract finite state models from execution logs. However, existing techniques do not scale well when processing very large logs, such as system-level logs obtained by combining component-level logs. Furthermore, in the case of component-based systems, existing techniques assume to know the definitions of communication channels between components. However, this detailed information is usually not available in the case of systems integrating 3rd-party components with limited documentation.

In this chapter, we address the scalability problem of inferring the model of a component-based system from the individual component-level logs, when the only available information about the system are high-level architecture dependencies among components and a (possibly incomplete) list of log message templates denoting communication events between components.

This chapter is organized as follows: Section 4.1 introduces the current state and limitations of model inference techniques. Section 4.2 illustrates the motivating example. Section 4.3 describes the different steps of the core algorithm of SCALER. Section 4.4 reports on the evaluation of SCALER. Section 4.5 concludes this chapter.

4.1 Overview

Behavior models of software system components play a key role in many software engineering tasks, such as program comprehension [26], test case generation [40], and model checking [22]. Unfortunately, such models either are scarce during software development or, if available, they quickly become outdated as the implementations evolve, because of the time and cost involved in generating and maintaining them [121].

One possible way to overcome the lack of software models is to use *model inference* techniques, which extract models—typically in the form of (some type of) Finite State Machine (FSM)—from execution logs. Although the problem of inferring a minimal FSM is NP-complete [13], there have been several proposals of polynomial-time approximation algorithms to infer FSMs [11, 13, 76] or richer variants, such as gFSM (guarded FSM) [80, 123] and gFSM extended with transition probabilities [37], to obtain more faithful models.

Although the aforementioned model inference techniques are fast and accurate enough for relatively small programs, all of them suffer from scalability issues, due to the intrinsic computational complexity of the problem. This leads to out-of-memory errors or extremely long, unpractical execution time when processing very large logs [125], such as system-level logs obtained by combining (e.g., through linearization) component-level logs. A recent proposal [76] addresses the scalability issue using a distributed FSM inference approach based on MapReduce. However, this approach requires to encode the data to be exchanged between mappers and reducers in the form of key-value pairs. Such encoding is application-specific; hence, it cannot be used in contexts—like the one in which this work has been performed—in which the system is treated as a black-box (i.e., the source code is not available), with limited information about the data recorded in the individual components logs.

Another limitation of state-of-the-art techniques is that they cannot infer, from component-level logs, a system-level model that captures both the individual behaviors of the system's components and the interactions among them. Such a scenario can be handled with existing model inference techniques for distributed systems, such as CSight [10], which typically assume the availability of channels definitions, i.e., the exact definition of which events communicate with each other between components. However, this information is not available in many practical contexts, where the system is composed of heterogenous, 3rd-party components, with limited documentation about the messages exchanged between components and the events recorded in logs.

In this chapter, we address the scalability problem of inferring the model of a component-based system from the individual component-level logs (possibly coming from multiple executions), when the only available information about the system are high-level architecture dependencies among components and a (possibly incomplete) list of log message templates denoting communication events between components. Our goal is to infer a system-level model that captures not only the components' behaviors reflected in the logs but also the interactions among them.

Our approach, called SCALER, follows a *divide and conquer* strategy: we first infer a model of each component from the corresponding logs using a state-of-the-art model inference technique, and then we "stitch" (i.e., we do a peculiar type of merge) the individual component models into a system-level model by taking into account the dependencies among the components, *as reflected in the logs*. The rationale behind this idea is that, though existing model inference techniques cannot deal with the size of all combined component logs, they can still be used to infer the models of individual components, since their logs are sufficiently small. In other words, SCALER tames the scalability issues of existing techniques by applying them on the smaller scope defined by component-level logs.

We implemented SCALER in a prototype tool, which uses MINT [123], a state-of-the-art technique for inferring gFSM, to infer the individual component-level models. We evaluate the scalability (in terms of execution time) and the accuracy (in terms of recall and specificity) of SCALER in comparison with MINT (fed with system-level logs reconstructed from component-level logs), on 7 proprietary datasets from one of our industrial partners in the satellite domain. The results show that our approach SCALER is about 245 times (on average) faster and can process larger logs than MINT. It generates nearly correct (with specificity always higher than 0.96) and largely complete models (with an average recall of 0.79), achieving higher recall than MINT (with a difference ranging between $+25\,pp$ and $+56\,pp$, with pp=percentage points) while retaining similar specificity.

To summarize, the main contributions of this chapter are:

- the SCALER approach for taming the scalability problem of inferring the model of a component-based system from the individual component-level logs, especially when only limited information about the system is available;

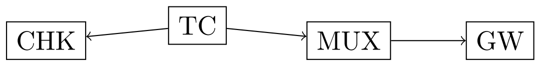

Figure 4.1: The components of the example system and their dependencies

- the empirical evaluation, in terms of scalability and accuracy, of SCALER and its comparison with a state-of-the-art approach.

4.2 Motivations

In this section, we discuss the motivations for this work using an example based on a real system from one of our industrial partners in the satellite domain. We consider a simplified version of a satellite ground control system, composed of the four components shown in Figure 4.1: *TC*, the module handling tele-commands for the satellite, which is also the entry point of the system; *MUX*, a multiplexer combining different tele-commands into a single communication stream; *CHK*, the module validating the tele-commands parameters before they are sent to the satellite; *GW*, the gateway managing the connections between the satellite and the ground control system. Figure 4.1 also shows the architectural dependencies among components; for example, the arrow from component *TC* to component *MUX* indicates that *TC uses* (or invokes) an operation provided by *MUX*. Every execution of the system generates a set of logs, with one log for each component; Figure 4.2 depicts the logs of the four system components generated in two executions; for space reasons, the format of timestamps has been compressed.

To infer a model from these individual component logs, one could use existing model inference techniques for distributed systems, such as CSight [10]. These techniques typically assume the availability of channels definitions, i.e., the exact definition of which events communicate to each other between components. However, this information is not available in many practical contexts, including ours, where the system is composed of heterogenous, 3rd-party components, with limited documentation. More specifically, the only available information about the system are high-level architecture dependencies among components (like those in Figure 4.1) and a (possibly incomplete) list of communication events, without knowing exactly how events communicate with each other. Due to this limited information, we cannot use existing techniques for model inference for distributed systems.

Another approach towards model inference would be to reconstruct a system-level log from the individual component logs and use non-distributed model

CMP	Execution 1	Execution 2
TC	$e^{TC}_{1,1}$= `14:26:01 sending X via f0`	$e^{TC}_{1,2}$= `14:30:11 sending Y via f1`
	$e^{TC}_{2,1}$= `14:26:02 TC accepted`	$e^{TC}_{2,2}$= `14:30:12 wait message`
MUX	$e^{MUX}_{1,1}$= `14:26:01 initialize`	$e^{MUX}_{1,2}$= `14:30:11 initialize`
	$e^{MUX}_{2,1}$= `14:26:01 commandName = X`	$e^{MUX}_{2,2}$= `14:30:12 commandName = Y`
	$e^{MUX}_{3,1}$= `14:26:01 commandName = X`	$e^{MUX}_{3,2}$= `14:30:12 data flow ID = f1`
	$e^{MUX}_{4,1}$= `14:26:01 data flow ID = f0`	$e^{MUX}_{4,2}$= `14:30:12 send = no`
	$e^{MUX}_{5,1}$= `14:26:02 send= ok`	
GW	$e^{GW}_{1,1}$= `14:26:01 encrypt TC_01`	$e^{GW}_{1,2}$= `14:30:12 reject command`
CHK	$e^{CHK}_{1,1}$= `14:26:01 mode 1`	$e^{CHK}_{1,2}$= `14:30:11 mode 0`
	$e^{CHK}_{2,1}$= `14:26:02 automatic config`	

Log Message Templates

$*tmp_1$= sending v_1 via v_2	$*tmp_2$= TC accepted	$*tmp_3$= wait message
$*tmp_4$= initialize	tmp_5= cmdName = v_1	tmp_6= data flow ID = v_1
$*tmp_7$= send = v_1	$*tmp_8$= encrypt v_1	$*tmp_9$= reject command
$*tmp_{10}$= mode v_1	$*tmp_{11}$= automatic config	

Figure 4.2: (top) Component logs generated by two executions of the example system; (bottom) Log message templates extracted from components logs (communication events are marked with an asterisk).

inference techniques such as MINT [123] or GK-tail+ [80]. However, such approaches typically suffer from scalability issues due to the underlying algorithms they use. For example, the main algorithm used in MINT has worst-case time complexity that is cubic in the size of the inferred model [70]; the algorithm used for removing non-determinism from models can exhibit, based on our preliminary evaluation, deep recursion that causes stack overflows and makes MINT crash. Furthermore, GK-tail+ is not publicly available and the largest log on which it was evaluated contained 11386 log entries. Since the system of our industrial partner can generate, when considering all the components, logs with more than 30000 entries, there is need for a scalable model inference technique that can process component logs.

4.3 Scalable Model Inference

Our technique for system model inference from component logs follows a *divide and conquer* approach. The idea is to first infer a model of each system component from the corresponding logs; then, the individual component models are

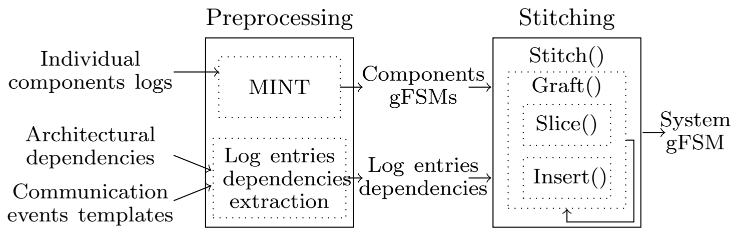
Preprocessing
Stitching
Individual components logs
Architectural dependencies
Communication events templates
MINT
Log entries dependencies extraction
Components gFSMs
Log entries dependencies
Stitch()
Graft()
Slice()
Insert()
System gFSM

Figure 4.3: Workflow of the SCALER technique

merged together taking into account the dependencies among components, *as reflected in the logs*. We call this process SCALER. The rationale behind our technique is that though existing (log-based) model inference techniques cannot deal with the size of all combined component logs, they can still be used to accurately infer the models of individual components, since their logs are sufficiently small for the existing model inference techniques to work. The challenge is then how to "stitch" together the models of the individual components to build a system model that reflects not only the components behavior but also their dependencies, while preserving the accuracy of the component models. For example, simply appending one component model after the other perfectly preserves the accuracy of the inferred component models, but it significantly loses the dependencies between components. On the other hand, performing a parallel composition of automata on the component models (based on the dependencies between components) loses the accuracy of the component models because of the over-generalization caused by the parallel composition. To solve this problem, we develop a set of novel algorithms that take into account the dependencies between components while preserving the component models as much as possible.

Figure 4.3 outlines the workflow of SCALER. The technique takes as input the logs of the different components, possibly coming from multiple executions, a description of the architectural dependencies among components, and a list of log message templates denoting communication events between components; it returns a system level gFSM. The main two stages of the SCALER technique are *pre-processing* and *stitching*. The *pre-processing* stage includes two steps:

- step 1 infers, for each component, its gFSM based on the corresponding logs;

- step 2 derives, using the architectural dependencies and the message templates of communicating events, the *log entries dependencies* of each execution.

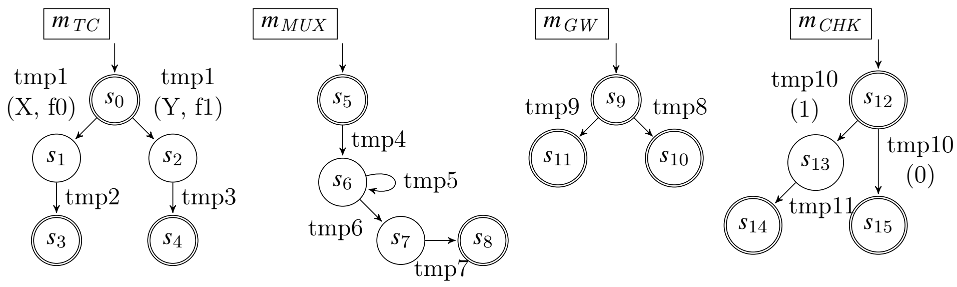

Figure 4.4: Component-level gFSMs inferred by MINT from the logs shown in Table 4.2

The intermediate outputs of the pre-processing step are then used in the *stitching* stage, which is at the core of our technique: in this stage, we "stitch" together the different component-level gFSMs, taking into account the log entries dependencies, to build a system-level gFSM. We describe these two stages in the following subsections.

4.3.1 Pre-processing Stage

4.3.1.1 Inferring Component Models

We infer component-level models using MINT [123], a state-of-the-art tool that is publicly available.

MINT takes as input (1) the logs produced by the individual component for which one wants to infer the model and (2) the templates of the events recorded in the component logs. The event templates are required to parse the log entries, to retrieve the actual events and their parameters. Nevertheless, often such templates are not available or documented. This situation is typical when dealing with 3rd-party, black-box components—as it is the case for the ground control system used by our industrial partner—and it is known in the literature as the log message format identification problem. We use MoLFI [84], a state-of-the-art solution for this problem, to derive the event templates that are then used by MINT; as an example, the box at the bottom of Figure 4.2 shows the templates produced by MoLFI from the logs of our running example.

The models inferred by MINT are gFSMs; Figure 4.4 shows the component-level gFSMs inferred by MINT for the four components of our running example. We use a compact notation for the guards on the event parameters labeling the guarded transitions; for example, in the gFSM of *TC* (i.e., m_{TC}), the guard $(\texttt{X}, \texttt{f0})$ stands for $(v_1 = \text{“}\texttt{X}\text{”}, v_2 = \text{“}\texttt{f0}\text{”})$.

4.3.1.2 Identifying Log Entries Dependencies

A system-level model of a component-based system has to capture not only the behavior of the individual components but also the intrinsic behavioral dependencies among them. For example, considering the fact that TC invokes MUX as shown in Figure 4.1, one could speculate that the event recorded in entry $e^{TC}_{1,1}$ could lead to the event recorded in entry $e^{MUX}_{1,1}$ in Figure 4.2; if this is the case, the model should reflect this dependency.

Component dependencies can be extracted from the source code by means of program analysis or from existing models such as UML Sequence Diagrams [16, 127]. However, such techniques require either access to the source code or the existence of complete documentation with fine-grained information. None of these conditions can be fulfilled in the common situation where systems integrate many 3rd-party components, either because the source code is not available or because software documentation is limited. This is the case for the example system provided by our industrial partner: the source code of 3rd-party components is not available, the architectural documentation only includes coarse-grained dependencies (like those shown in Figure 4.1), and the only additional information is the knowledge of domain experts, who can provide an incomplete list of log message templates corresponding to events related to the "interactions" between components (e.g., "send" and "receive" events). We remark that this list of message templates, which we call *communication events* and are recorded in *communication log entries*, cannot be used to define communication channels since we do not know *how* these events are used for communication between components. For all these reasons, we need to extract additional information, in the form of more fine-grained dependencies that reflect the logged events and their timestamps; we call such dependencies *log entries dependencies*.

The idea at the basis of our log entries dependencies extraction process is that, if there is an architectural dependency from component c_X to another component c_Y (representing the use of c_Y by c_X), then there is at least (an event recorded in) a log entry of c_Y that is the consequence of (an event recorded in) a log entry of c_X. This is because we can partition the log entries of c_Y in two disjoint classes: (1) *externally generated ("ext-gen")*, containing log entries that are produced as the consequence of a communication (event recorded in a) log entry of c_X; (2) *internally generated ("int-gen")*, all the other log entries, i.e., those immediately following either an "ext-gen" log entry or another "int-gen" entry. This means that to extract the dependencies of all log entries between c_X and c_Y, we first have to identify the "ext-gen" log entries of c_Y.

Table 4.1: Extracted log entries dependencies for the running example

Execution	**Log entry dependencies**
Exec1	$e_1^{TC} \rightsquigarrow \langle e_1^{MUX}, e_2^{MUX}, e_3^{MUX}, e_4^{MUX} \rangle$ $e_1^{TC} \rightsquigarrow \langle e_1^{CHK} \rangle$, $e_2^{TC} \rightsquigarrow \langle e_5^{MUX} \rangle$ $e_2^{TC} \rightsquigarrow \langle e_2^{CHK} \rangle$, $e_4^{MUX} \rightsquigarrow \langle e_1^{GW} \rangle$
Exec2	$e_1^{TC} \rightsquigarrow \langle e_1^{MUX}, e_2^{MUX}, e_3^{MUX} \rangle$ $e_1^{TC} \rightsquigarrow \langle e_1^{CHK} \rangle$, $e_2^{TC} \rightsquigarrow \langle e_4^{MUX} \rangle$ $e_4^{MUX} \rightsquigarrow \langle e_1^{GW} \rangle$

The identification of the "ext-gen" log entries of c_Y is based on an intuitive observation: a communication log entry e_y of a component c_Y is produced as the consequence of a communication log entry e_x of a component c_X *only if* there is an architectural dependency between c_X and c_Y, and the timestamp of e_x is less than or equal[1] to the timestamp of e_y. However, by using only this observation, we cannot determine the correct pair (e_x, e_y) of communication log entries if there are multiple candidate pairs that satisfy the same constraint on the timestamp. To illustrate this case, let us consider communication log entries $\langle e_1^X, e_2^X \rangle$ of c_X and $\langle e_1^Y, e_2^Y \rangle$ of c_Y, where the timestamp of e_1^X is t_1, the timestamps of both e_2^X and e_1^Y are t_2, and the timestamp of e_2^Y is t_3, with $t_1 < t_2 < t_3$. There are three candidate pairs of communication log entries that satisfy the constraint on the timestamp: (e_1^X, e_1^Y), (e_2^X, e_1^Y), and (e_2^X, e_2^Y). In such a case, we use an heuristic and select the pair with the smallest timestamp difference; in the current example, we would select (e_2^X, e_1^Y) and say that e_2^X *leads-to (inter-component)* e_1^Y, denoted with $e_2^X \rightsquigarrow_E e_1^Y$, to represent the inter-component communication dependency. In our running example, given the list of templates corresponding to (log entries of) communication events: tmp_1, tmp_2, tmp_4, and tmp_7, if we consider the architectural dependency from TC to MUX and focus on the first execution, we say that $e_1^{TC} \rightsquigarrow_E e_1^{MUX}$ and $e_2^{TC} \rightsquigarrow_E e_5^{MUX}$.

We remark that our heuristic may introduce some imprecisions with logs in which the timestamp granularity is relatively coarse-grained (e.g., seconds instead of milli- or nano-seconds) and the communication between components is fast enough such that often two communication events that logically occur one before the other are logged using the same timestamp; in such cases, there

[1]We assume that the clocks of the different components are synchronized, for example using the Network Time Protocol (NTP) [87].

would still be multiple candidate pairs[2].

After identifying the "ext-gen" log entries of c_Y, every "int-gen" log entry of c_Y can be related to the most recent "ext-gen" log entry of c_Y. More precisely, if we have a log $\langle \dots, e_{i_0}, e_{i_1}, \dots, e_{i_n}, \dots \rangle$ of c_Y where e_{i_0} is an "ext-gen" log entry followed by the sequence of "int-gen" log entries $\langle e_{i_1}, \dots, e_{i_n} \rangle$, we say that e_{i_0} *leads-to (intra-component)* e_{i_j}, denoted with $e_{i_0} \rightsquigarrow_I e_{i_j}$, for $j \in \{1, \dots, n\}$. In our running example, when considering the logs of *TC* and *MUX* in the first execution, given the "ext-gen" log entries e_1^{MUX} and e_5^{MUX} identified as above, we say that $e_1^{MUX} \rightsquigarrow_I e_j^{MUX}$ for $j \in \{2, 3, 4\}$.

By composing the *leads-to (inter-component)* and the *leads-to (intra-component)* relations, we can finally extract the log entries dependencies. More precisely, if we have a log $\langle \dots, e_k^X, \dots \rangle$ of c_X and a log $\langle \dots, e_{i_0}^Y, e_{i_1}^Y, \dots, e_{i_n}^Y, \dots \rangle$ of c_Y, such that $e_k^X \rightsquigarrow_E e_{i_0}^Y$ and $e_{i_0}^Y \rightsquigarrow_I e_{i_j}^Y$ for $j \in \{1, \dots, n\}$, we say that e_k^X *leads-to* $\langle e_{i_0}^Y, e_{i_1}^Y, \dots, e_{i_n}^Y \rangle$, denoted with $e_k^X \rightsquigarrow \langle e_{i_0}^Y, e_{i_1}^Y, \dots, e_{i_n}^Y \rangle$. When considering *TC* and *MUX* in the first execution of our running example, we have $e_1^{TC} \rightsquigarrow \langle e_1^{MUX}, e_2^{MUX}, e_3^{MUX}, e_4^{MUX} \rangle$ because $e_1^{TC} \rightsquigarrow_E e_1^{MUX}$ and $e_1^{MUX} \rightsquigarrow_I e_j^{MUX}$ for $j \in \{2, 3, 4\}$ as identified above; also, we have $e_2^{TC} \rightsquigarrow \langle e_5^{MUX} \rangle$ because $e_2^{TC} \rightsquigarrow_E e_5^{MUX}$ and e_5^{MUX} does not *lead-to (intra-component)* anything. Table 4.1 shows all the log entries dependencies extracted for the log entries in Figure 4.2.

4.3.2 Stitching Stage

In this stage, we build a system-level gFSM that captures not only the components' behavior inferred from the logs but also their dependencies as reflected in the log entries dependencies identified in the pre-processing stage.

Since the dependencies between components observed through the logs are different from execution to execution, we first build system-level gFSM *for each execution* and then merge these gFSMs together using the standard DFA (Deterministic Finite Automaton) union operation[3]. We call this process "stitching" whereas we call "grafting" the inner process that builds a system-level gFSM for each execution. The pseudocode of the top-level process Stitch is shown in Algorithm 2.

We assume that, within a set of components C, there is a component labeled c_{main} that corresponds to the root component in the system architectural

[2]Multiple candidate pairs could be addressed by exploring all potential log entries dependencies for the construction of different models; we leave this as part of future work.

[3]MINT produces a deterministic gFSM $m = (S, ET, G, \delta, s_0, F)$, with $\delta : S \times ET \times G \to S$; it can be easily converted into a DFA $m' = (S, \Sigma, \delta', s_0, F)$ with $\delta' : S \times \Sigma \to S$ where $\Sigma = ET \times G$.

Algorithm 2 STITCH

```
Input: Set of Components $C = \{c_{main}, c_1, \ldots, c_n\}$
       Set of gFSMs $M = \{m_{c_{main}}, m_{c_1}, \ldots, m_{c_n}\}$
       Set of Logs $L_{main} = \{l_1, \ldots, l_k\}$
Output: System model $m_{sys}$
1: Set of gFSMs $W \leftarrow \emptyset$
2: for each $l_i \in L_{main}$ do
3:     gFSM $m_{main} \leftarrow$ GRAFT$(c_{main}, l_i, M)$
4:     $W \leftarrow \{m_{main}\} \cup W$
5: end for
6: gFSM $m_{sys} \leftarrow DFAUnion(W)$
7: return $m_{sys}$
```

diagram (e.g., *TC* in our running example). Algorithm STITCH takes as input C, a set of component-level gFSMs M (one model for each component in C), and a set of logs (one log for each execution) L_{main} for c_{main}; it returns a system-level gFSM m_{sys}. Internally, STITCH uses novel auxiliary algorithms (GRAFT, SLICE, INSERT), which are described further below.

The algorithm builds a system-level gFSM m_{main} for each execution log $l_i \in L_{main}$, starting from the component-level gFSMs in M (lines 1–4); this is done by the GRAFT algorithm, described in detail in § 4.3.2.1. During the iteration through the execution logs in L_{main}, the resulting system-level gFSMs m_{main} are collected in the set W. Last, the gFSMs in W are merged into m_{sys} using the DFA union operation[4] (line 6). The algorithm ends by returning the system-level gFSM m_{sys} (line 7), inferred from all executions in L_{main}.

4.3.2.1 Graft

The GRAFT algorithm builds the system-level gFSM for an execution by merging the individual component-level gFSMs, taking into account the log entries dependencies extracted from the execution, while preserving the component gFSMs as much as possible. To illustrate the main idea behind the algorithm, let us consider two components c_X and c_Y, whose corresponding gFSMs (inferred in the pre-processing stage) m_{c_X} and m_{c_Y} are shown in Figure 4.5. These

[4]One could use the standard DFA minimization after the DFA union in line 6 to reduce the size of the system-level gFSM. However, our preliminary evaluation showed that the minimization operation can reduce the gFSM size (in terms of numbers of states and transitions) by at most 5%, and it increases the execution time of the STITCH algorithm by more than five times.

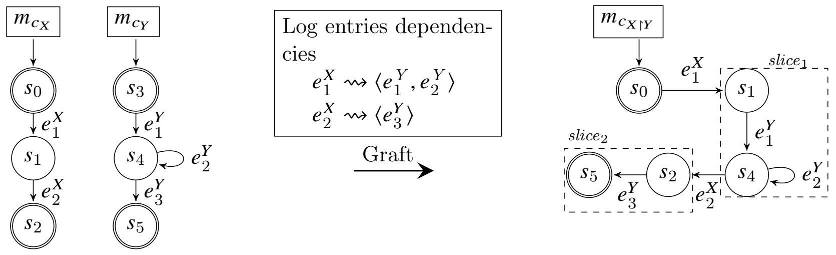

Figure 4.5: The main intuition behind the Graft algorithm (for simplicity, we use log entries as transition labels)

gFSMs respectively accept log $l_X = \langle e_1^X, e_2^X \rangle$ and log $l_Y = \langle e_1^Y, e_2^Y, e_3^Y \rangle$. Let us also assume that in terms of log entries dependencies (expressed through the *leads-to* relation) we have $e_1^X \rightsquigarrow \langle e_1^Y, e_2^Y \rangle$ and $e_2^X \rightsquigarrow \langle e_3^Y \rangle$. Taking into account these dependencies, intuitively we can say that the gFSM resulting from the merge of m_{c_X} and m_{c_Y}, denoted by $m_{c_{X \upharpoonright Y}}$, should accept the sequence of log entries $\langle e_1^X, e_1^Y, e_2^Y, e_2^X, e_3^Y \rangle$. To obtain $m_{c_{X \upharpoonright Y}}$, we first "slice" m_{c_Y} into two gFSMs: $slice_1$ (accepting $\langle e_1^Y, e_2^Y \rangle$) and $slice_2$ (accepting $\langle e_3^Y \rangle$); then, we "insert" 1) $slice_1$ as the target of the transition of m_{c_X} that reads e_1^X, and 2) $slice_2$ as the target of the transition of m_{c_X} that reads e_2^X. Note that the self-loop transition in m_{c_Y} is preserved in $m_{c_{X \upharpoonright Y}}$ as a result.

Algorithm 3 shows the pseudocode of the Graft algorithm. The algorithm takes as input a component c_{cur}, an execution log $l_{cur} = \langle e_1, \ldots, e_z \rangle$, and a set of component-level gFSMs $M = \{m_{c_{main}}, m_{c_1}, \ldots, m_{c_n}\}$; it returns a gFSM m_{sl} that accepts the sequence of log entries composed of the entries $e_i \in l_{cur}$, with each e_i interleaved with the log entries to which it *leads-to*.

The algorithm starts by slicing the gFSM $m_{c_{cur}}$ of the input component c_{cur} into a gFSM m_{sl} that accepts only l_{cur} (line 2); the actual slicing is done through algorithm Slice, described in detail in § 4.3.2.2. The rest of the algorithm expands m_{sl} taking into account the log entries dependencies (lines 3–14): for each log entry $e_i \in l_{cur}$, a gFSM m_g that accepts the log entries sequence that e_i *leads-to* is built and "inserted" in m_{sl} as the target of the guarded transition gt that reads e_i. More precisely, the algorithm performs a run of m_{sl} as if it were to accept the log l_{cur}: starting from the initial state of m_{sl} (line 3), it moves to the next state s by making the guarded transition gt that reads e_i (line 5). As part of this move, for each log entry sequence l_d such that $e_i \rightsquigarrow l_d$, we recursively call Graft to build the gFSM m_g that accepts l_d; this gFSM is then added to the set W (lines 7–10) . Since a log entry e_i may *lead-to* log entries sequences of multiple components, we compose the

Algorithm 3 Graft

```
Input: Component $c_{cur}$
       Log $l_{cur} = \langle e_1, \dots, e_z \rangle$
       Set of gFSMs $M = \{m_{c_{main}}, m_{c_1}, \dots, m_{c_n}\}$
Output: System model for the current execution $m_{sl}$
 1: gFSM $m_{cur} \leftarrow getComponentGFSM(M, c_{cur})$
 2: gFSM $m_{sl} \leftarrow$ SLICE($m_{c_{cur}}, l_{cur}$)
 3: State $s \leftarrow getInitialState(m_{sl})$
 4: for each $e_i \in l_{cur}$ do
 5:     GuardedTransition $gt \leftarrow getGuardedTran(m_{sl}, s, e_i)$
 6:     Set of gFSMs $W \leftarrow \emptyset$
 7:     for each log entries sequence $l_d \mid e_i \rightsquigarrow l_d$ do
 8:         Component $c_d \leftarrow getComponentFromLog(L_d)$
 9:         gFSM $m_g \leftarrow$ GRAFT($c_d, L_d, M$)
10:         $W \leftarrow \{m_g\} \cup W$
11:     end for
12:     gFSM $m_{pl} \leftarrow DFAParallelComposition(W)$
13:     $m_{sl} \leftarrow$ INSERT($m_{sl}, gt, m_{pl}$)
14:     $s \leftarrow getTargetState(gt)$
15: end for
16: return $m_{sl}$
```

individual gFSMs in W using the standard DFA parallel composition operation (line 12). The resulting gFSM m_{pl} is "inserted" in m_{sl} as the target of gt by the INSERT algorithm (line 13), described in detail in § 4.3.2.3. At the end of each iteration of the loop, the state s is updated with the target state of the gt transition (line 14).

As an example, let us consider the case in which the STITCH algorithm calls the GRAFT algorithm when processing Execution-2 of our running example. Figure 4.6-(a) shows the component-level gFSM and how they are related when taking into account the *leads-to* relation listed in Table 4.1. Algorithm STITCH invokes GRAFT with parameters $c_{cur} = TC$, $l_{cur} = \langle e^{TC}_{1,2}, e^{TC}_{2,2} \rangle$, $M = \{m_{TC}, m_{MUX}, m_{CHK}, m_{GW}\}$. The call to SLICE yields the gFSM *slice₁* shown in Figure 4.6-(a); it accepts $\langle e^{TC}_{1,2}, e^{TC}_{2,2} \rangle$, using the transitions labeled with $tmp_1(Y, f1)$ and tmp_3. Then, starting from s_0 of $slice_1$, the invocation of the auxiliary function *getGuardedTran* yields the guarded transition $(s_0, tmp_1, [Y, f1], s_2)$ that reads $e^{TC}_{1,2}$. Since $e^{TC}_{1,2} \rightsquigarrow \langle e^{MUX}_{1,2}, e^{MUX}_{2,2}, e^{MUX}_{3,2} \rangle$ and $e^{TC}_{1,2} \rightsquigarrow e^{CHK}_{1,2}$, the algorithm makes a recursive call for $\langle e^{MUX}_{1,2}, e^{MUX}_{2,2}, e^{MUX}_{3,2} \rangle$, which returns the sliced gFSM $slice_2$, and for $\langle e^{CHK}_{1,2} \rangle$, which returns $slice_3$;

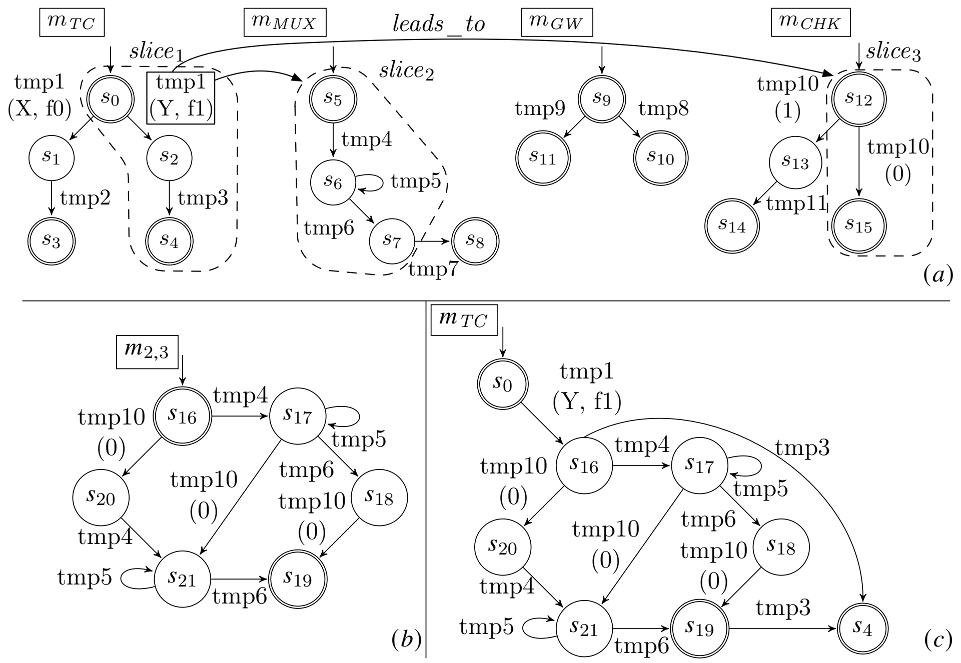

Figure 4.6: Application of algorithm Graft to Execution-2 of the running example

both gFSMs are shown in Figure 4.6-(a). At the end of the inner loop, we have $W = \{slice_2, slice_3\}$; their parallel composition is $m_{2,3}$ and is shown in Figure 4.6-(b). This gFSM is then inserted in $slice_1$ as the target of the transition $(s_0, tmp_1, [Y, f1], s_2)$, as shown in Figure 4.6-(c). The algorithm ends for $e^{TC}_{1,2}$ by inserting $m_{2,3}$ in s_2 and moves on to the next log entry $e^{TC}_{2,2}$.

4.3.2.2 Slice

This algorithm takes as input a component-level gFSM m_c and a log l_c; it returns a new gFSM m_{sl}, which is the sliced version of m_c and accepts only l_c.

Its pseudocode is shown in Algorithm 4. First, the algorithm retrieves the state of m_c that will become the initial state s of the sliced gFSM m_{sl} (line 2). Upon the first invocation of Slice for a certain gFSM m_c, s will be the initial state of m_c; for the subsequent invocations, s will be the last state visited in m_c when running the previous slice operations. Starting from s, the algorithm performs a run of m_c as if it were to accept the log l_c: the traversed states and guarded transitions of m_c are added into m_{sl} (lines 3–6). At the end of

Algorithm 4 Slice

```
Input: A component gFSM $m_c$
       A component Log $l_c = \langle e_1, \ldots, e_z \rangle$
Output: a sliced gFSM $m_{sl}$
1: gFSM $m_{sl} \leftarrow \mathit{initGFSM}()$
2: State $s \leftarrow \mathit{getSliceStartState}(m_c)$
3: for each $e_i \in l_c$ do
4:     Guarded Transition $gt \leftarrow \mathit{getGuardedTran}(m_c, s, e_i)$
5:     $m_{sl} \leftarrow \mathit{AddGuardedTranAndStates}(m_{sl}, gt)$
6:     $s \leftarrow \mathit{getTargetState}(gt)$
7: end for
8: $\mathit{updateSliceStartState}(m_c, s)$
9: return $m_{sl}$
```

the loop, the algorithm records (line 8) the last state visited in m_c when doing the slicing, which will be used as the initial state of the next slice on m_c; it then ends by returning m_{sl}.

4.3.2.3 Insert

We recall that this algorithm is invoked by the Graft algorithm to "insert" a gFSM m_y into a gFSM m_x as the target of a guarded transition gt of m_x, taking into account the log entries dependencies. More specifically, let us consider a log entry e and a set of logs $L = \{l_1, \ldots, l_n\}$ where $e \rightsquigarrow l_i$ for $i = 1, \ldots, n$; the transition gt of m_x reads e, and m_y is the parallel composition of the gFSMs that accepts the logs in L. The Insert algorithm merges m_y into m_x such that, by "inserting" m_y as the target of the guarded transition gt, m_x can read the (entries in the) logs in L right after reading e.

We illustrate how the algorithm works through the example in Figure 4.7, in which the input gFSMs m_x and m_y are shown on the left side; we will insert m_y into m_x as the target of the guarded transition gt, labeled with a and having s_t as target state. Without loss of generality, we assume that m_y has only one transition (labeled with α) between its initial state s_i and the final one s_f. The main idea behind the Insert algorithm is to duplicate both incoming and outgoing transitions of the target state of gt, and to redirect the new copies to the initial and finals states of m_y. More specifically:

- the incoming transition gt of s_t (labeled with a) is duplicated and the new copy is redirected, by changing its target state, to the initial state of m_y (i.e., s_i);

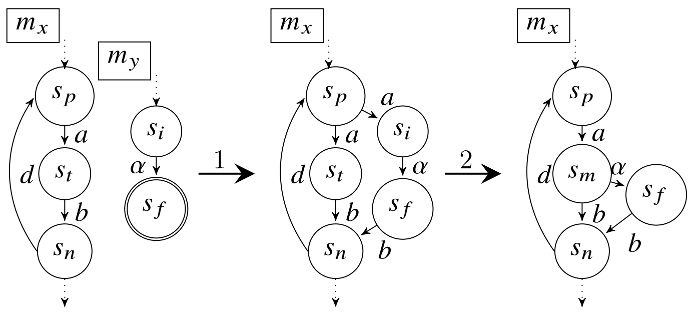

Figure 4.7: Example showing the basic idea of the Insert algorithm, when inserting m_y into m_x as the target of the guarded transition gt with $gt = (s_p, a, s_t)$. Step 1 shows the application of duplication and redirection; step 2 applies determinization to merge states s_t and s_i.

- the outgoing transitions of s_t (e.g., the one labeled with b) are duplicated and the new copies are redirected, by changing the source state, such that they originate from the final state of m_Y (i.e., s_f).

The updated m_x, resulting from the application of duplication and redirection, is shown in the middle of Figure 4.7. We remark that we keep the original incoming and outgoing transitions of s_t on purpose, to take into account the cases in which one of the log entries read by gt does not *lead-to* log entries read by the transition labeled with α. Duplication and redirection operations introduce some nondeterminism in m_x; in our example, s_p has two outgoing transitions both labeled with a. We remove nondeterminism using a *determinization* procedure [27], which recursively merges pair of states that introduces nondeterminism[5]; in our example, the determinization procedure will merge s_t and s_i. The final m_x is shown on the right side of Figure 4.7.

Algorithm 5 shows the pseudocode of the Insert algorithm. The algorithm takes a gFSM m_x, a guarded transition gt, and a gFSM m_y; it returns the updated m_x that includes m_y as the target of gt. In the algorithm, s_t is the target state of gt, s_i is the initial state of m_y and F_y is the set of the final states of m_y. The core part (lines 4–10) iterates through each guarded transition t of s_t, duplicates it, and redirects the new copy as described above, using the the auxiliary function *duplicateAndRedirectTransitions*. Last, the algorithm removes nondeterminism using *determinize* (line 11); it ends by returning the updated gFSM m_x (line 12).

[5]This procedure is different from the standard NFA (non-deterministic finite automaton) to DFA conversion since it yields an automaton which may accept a more general language than the NFA it starts from [27].

Algorithm 5 Insert

```
Input: gFSM $m_x$
       Guarded Transition $gt$
       gFSM $m_y$
Output: Updated gFSM $m_x$
 1: State $s_t \leftarrow getTargetState(gt)$
 2: State $s_i \leftarrow getInitialState(m_y)$
 3: Set of States $F_y \leftarrow getFinalStates(m_y)$
 4: for each Guarded Transition $t$ of $s_t$ do
 5:     if $t = gt$ then
 6:         $duplicateAndRedirectTransitions(t, s_t, \{s_i\})$
 7:     else if $t$ is an outgoing transition then
 8:         $duplicateAndRedirectTransitions(t, s_x, F_y)$
 9:     end if
10: end for
11: $determinization(m_x)$
12: return $m_x$
```

Accuracy of the system-level gFSM SCALER has three main sources of over-generalization that reduce the accuracy: (1) component-level model inference, (2) parallel composition in GRAFT, and (3) determinization in INSERT. The first is essentially inevitable in any model inference algorithm; we try to compensate it by using a state-of-the-art tool (MINT) to infer component models that are as accurate as possible. The second source may become a problem when the log dependencies identified in the preprocessing stage are incorrect; nevertheless, over-generalization caused by parallel composition is limited because the latter is only performed on the sliced gFSMs. The last source has limited effects because recursive determinization rarely occurs in practice.

We further discuss the accuracy of SCALER corroborated by experimental data in the next section.

4.4 Evaluation

We have implemented the SCALER approach as a Python program. In this section, we report on the evaluation of the performance of the SCALER implementation in generating the model of a component-based system from the individual component-level logs.

First, we assess the scalability of SCALER in inferring models from large execution logs. This is the primary dimension we focus on since we propose SCALER as a viable alternative to state-of-the-art techniques for processing large logs. Second, we analyze how accurate the models generated by SCALER are. This is an important aspect because it is orthogonal to scalability and has direct implications on the possibility of using the models generated by SCALER in other software engineering tasks (e.g., test case generation). Summing up, we investigate the following research questions:

RQ1: *How scalable is SCALER when compared to state-of-the-art model inference techniques?*

RQ2: *How accurate are the models (in the form of gFSMs) generated by SCALER when compared to those generated by state-of-the-art model inference techniques?*

4.4.1 Benchmark and Evaluation Settings

We used a benchmark composed of industrial, proprietary datasets provided by one of our industrial partners, active in the satellite industry. The benchmark contains component-level logs recorded during the execution of a satellite ground control system, which includes six major components. We created the benchmark as follows. First, we executed system-level tests on the ground control system 120 times and, in each test execution, we collected the log files of the six major components. Then, we created seven datasets of size ranging from 5K to 35K, where the size is expressed in terms of the total number of log entries. We assembled each dataset by randomly selecting a number of executions out of the pool of 120 executions, such that the total size of the logs contained in the dataset matched the desired dataset size. By construction, each dataset contains logs of the six major components of the system. The first three columns of Table 4.2 show, for each dataset in our benchmark, the size and the number of executions included in it. In total, there are 92 unique templates (i.e., unique number of events) for all logs. The experiments have been executed on a high-performance computing platform, using one of its quad-core nodes running CentOS 7 on a 2.4GHz Intel Xeon E5-2680 v4 processor with 4G memory.

4.4.2 Scalability

4.4.2.1 Methodology

To answer RQ1, we assess the scalability of SCALER, in terms of execution time with respect to the size of the logs, in comparison with MINT [123], a state-of-the-art model inference tool. We selected MINT as baseline because other tools are either not publicly available or require information not available in most practical contexts, including ours (e.g., channels' definitions; see section 5.2).

We ran both tools to infer a system-level model for each dataset in our benchmark. We provided as input to SCALER 1) the logs of the six components recorded in the executions contained in each dataset; 2) the architectural dependencies among components; 3) the list of log message templates for communication events, received from a domain expert. As for MINT, we provided as input the system-level logs of the system executions contained in each dataset. We derived these system-level logs by linearizing the individual component logs in each execution, taking into account the log entries dependencies. To guarantee a fair comparison, these dependencies are the same as those extracted in the pre-processing stage of SCALER. Since the total number of possible system-level logs is extremely large due to the linearization of the parallel behaviors of the components, we only considered one system-level log for each execution.

We remark that we used *two* instances of MINT: the one used internally by SCALER to generate component-level models; the other one for the comparison in inferring system-level models. For both instances, we used the default configuration (i.e., state merging threshold $k = 2$ and J48 as data classifier algorithm) [123]. Furthermore, to identify the event templates required by the MINT instances to parse the log entries, we first used a state-of-the-art tool (MoLFI [84]) to compute them and then we asked a domain expert to further refine them, e.g., by collapsing similar templates into a single one. To take into account the randomness of the log linearization (i.e., only one linearized system-level log) for each execution of MINT, we ran both MINT and SCALER ten times on each dataset. For each run, we set an overall time out of 24h for the model inference process both for MINT and for SCALER.

To assess the statistical significance of the difference between the execution time of SCALER and MINT (if any), we used the non-parametric Wilcoxon rank sum test with a level of significance $\alpha = 0.05$. Furthermore, we used the Vargha-Delaney ($\hat{A}_{12}$) statistic for determining the effect size of the difference. In our case, $\hat{A}_{12} < 0.5$ indicates that the execution time of SCALER is lower than that of MINT.

4.4.2.2 Results

The columns under the header "Scalability" of Table 4.2 show the scalability results for SCALER and MINT. More precisely, column *MINT* indicates the

Table 4.2: Execution time (in seconds), recall, and specificity of SCALER and MINT

Dataset	Size	Exec	System-level gFSM						Scalability					Accuracy					
			States			Transitions			MINT	SCALER			SpeedUp	Recall			Specificity		
			MINT	SCALER	Ratio	MINT	SCALER	Ratio		Prep(s)	Stitch(s)	Total(s)		MINT	SCALER	$\Delta_{\mathrm{R}}(pp)$	MINT	SCALER	$\Delta_{\mathrm{S}}(pp)$
D05K	5058	13	617.7	3816	6.2	837.2	7295	8.7	319.6	6.0	5.8	11.8	27.1	0.09	0.65	56	1.00	0.98	-2
D10K	10208	28	1207.3	6647	5.5	1585.4	12720	8.0	2597.0	10.9	15.3	26.2	99.2	0.16	0.63	47	0.99	0.98	-1
D15K	15078	42	1582.6	4184	2.6	2064.4	8868	4.3	7403.8	13.4	19.8	33.2	222.8	0.52	0.82	30	0.99	0.97	-2
D20K	20094	56	2257.0	7463	3.3	2914.7	15851	5.4	16022.2	18.6	32.1	50.7	315.9	0.58	0.86	28	0.98	0.97	-1
D25K	25034	71	3067.2	9496	3.1	3976.3	18767	4.7	35378.6	24.3	58.2	82.5	428.7	0.56	0.83	27	0.98	0.96	-2
D30K	30103	86	2871.3	19467	6.8	3701.8	40204	10.9	59222.7	29.0	129.6	158.6	373.3	0.61	0.86	25	0.98	0.96	-2
D35K	35079	101	N/A	10432	N/A	N/A	19707	N/A	timeout	32.5	72	104.5	N/A	N/A	0.88	N/A	N/A	0.97	N/A
Avg	20093.4	56.7	1933.9	8786.4	4.6	2513.3	17630.3	7.0	20157.3	19.3	47.5	66.8	244.5	0.42	0.79	35.5	0.99	0.97	-1.67

execution time of MINT; columns *Prep*, *Stitch*, and *Total* indicate the average (over the ten runs) execution time (in seconds) and the corresponding standard deviation of SCALER for the pre-processing stage, the stitching stage, and the cumulative execution time, respectively; column *SpeedUp* reports the speedup of SCALER over MINT computed as $\frac{Time_{MINT}}{Time_{SCALER}}$ [102].

SCALER is faster than MINT for all the datasets in our benchmark; the speed-up ranges between 27x (for the dataset D05K) and 428x (for the dataset D25K). The speed-up increases with the size of the datasets and, thus, the benefit of using SCALER over MINT increases for larger logs. Note that MINT reached the time out for the largest dataset (D35K) without producing any model. The Wilcoxon test also confirms that the differences in execution time between SCALER and MINT are statistically significant (p-value < 0.01 for all datasets) and the Vargha-Delaney statistic indicates that the effect size is always *large* ($\hat{A}_{12} < 0.10$) for all datasets.

Analyzing the performance of the two instances of MINT, we can say that when MINT is used for component-level model inference is much faster than MINT used for system-level model inference because (1) the component logs are smaller than the system-level logs and (2) there is a higher similarity among component logs than system-level logs.

4.4.3 Accuracy

4.4.3.1 Methodology

To answer RQ2, we ran both MINT and SCALER to evaluate and compare their accuracy for each dataset, in terms of recall and specificity of the inferred models following previous studies [37, 80, 123]. Recall measures the ability of the inferred models of a system to accept "positive" logs; specificity measures the ability of the inferred models to reject "negative" logs. We computed these metrics by using the well-known k-folds cross validation method, which has also been used in previous work [37, 80, 123] in the area of model inference. This method randomly partitions a set of logs into k non-overlapping folds: $k-1$ folds are used as input of the model inference tool, while the remaining fold is used as "test set", to check whether the model inferred by the tool accepts the logs in the fold. The procedure is repeated k times until all folds have been considered exactly once as the test set. For each fold, if the inferred model successfully accepts a positive log in the test set, the positive log is classified as True Positive (TP); otherwise, the positive log is classified as False Negative (FN). Similarly, if an inferred model successfully rejects a negative log in the test set, the negative log is classified as True Negative (TN); otherwise, the negative log is classified as False Positive (FP). Based on the classification results, we calculated the recall (R) as $R = \frac{|TP|}{|TP|+|FN|}$, and the specificity (S) as $S = \frac{|TN|}{|TN|+|FP|}$.

As done in previous work [37, 80, 123], we synthesized negative logs from positive logs by introducing small changes (mutations): 1) swapping two randomly selected log entries, 2) deleting a randomly selected log entry, and 3) adding a log entry randomly selected from other executions. To make sure a log resulting from a mutation contains invalid behaviors of the system, we checked whether the sequence of entries around the mutation location (i.e., the mutated entries and the entries immediately before and after the mutants) did not also appear in the positive logs.

Note that we needed to derive system-level logs from the individual component logs in test sets to check the acceptance of the system-level models inferred by SCALER and MINT. To this end, as done for the scalability evaluation, for each execution in the test sets, we linearized the individual component logs to derive the system-level log. Also, to take into account the randomness of the derivation of system-level logs, we repeat the 10-folds cross validation ten times on each dataset and then applied statistical tests as done for the scalability evaluation.

4.4.3.2 Results

The columns under the header "Accuracy" of Table 4.2 show the results of MINT and SCALER in terms of recall, specificity, and difference of these values (in percentage points, pp) between SCALER and MINT.

MINT achieves high specificity scores, always greater than 0.98. However, recall is low, ranging between 0.09 for the D05K dataset and 0.61 for the D30K dataset. Notice that no results were obtained for the larger dataset with 35K log entries because MINT reached the timeout of 24h without generating any model. SCALER achieves a slightly lower specificity than MINT, with an average difference of 1.67pp. However, SCALER achieves substantially higher recall than MINT. The difference in recall values ranges between $+25pp$ (D30K dataset) and $+56pp$ (D05K dataset), with an average improvement of 35.5pp. Such a result can be explained mainly because MINT takes as input only one system-level log among all possible instances of the linearization of the parallel behaviors of the components for each execution, and fails to scale up to take as input all the possible system-level logs. Related to this, since SCALER takes as input all the possible system-level logs (in the form of component-level logs with the log entries dependencies) for each execution, it returns as output a system-level gFSM having on average 4.6x more states and 7.0x more transitions than MINT (see the columns under the header "System-level gFSM" in Table 4.2).

According to the Wilcoxon test, SCALER always achieves a statistically higher recall than MINT for all datasets (p-value < 0.01) with a large effect size. However, SCALER achieves a statistically lower specificity than MINT in five out of seven datasets (i.e., with 5K, 10K, 20K and 30K log entries). While the difference in specificity are statistically significant, it is worth noting that the magnitude of the difference is small, being no larger than 2pp.

4.4.4 Discussion and Threats to Validity

From the results above, we conclude that, for the large logs typically encountered in practice, SCALER provides results that are good enough to generate nearly correct (with a specificity always greater than 0.96) and largely complete models (with an average recall of 0.79).

The incompleteness of the inferred models is due to the limited knowledge we have on the system (i.e., the incomplete list of message templates characterizing communication events) and to the heuristic used in computing log entries dependencies, which is affected by the coarse-grained timestamp granularity of the logs included in our benchmark. In contrast, MINT, when used as a stand-alone tool on the same large logs, does not scale and fares poorly in terms of recall, generating very incomplete models.

From a practical perspective, the results achieved by SCALER lead to a considerable reduction of false negatives, with a marginal increment of false positives. For example, for the D15K dataset, MINT generates (in about two hours) a gFSM that accepts only 52% of the true positives (positive logs). In this case, engineers need to substantially modify the inferred gFSM to accept the remaining 48% of positive logs. Instead, for the same dataset, SCALER generates in about 33 seconds a gFSM that accepts 82% of the positive logs (and rejects 97% of the negative logs). The marginal decrement of the negative logs correctly dismissed by the gFSM inferred by SCALER is largely compensated by (1) a significant reduction of the number of wrongly rejected positive logs ($+30pp$ in recall), and (2) a substantial reduction of the execution time (SCALER is about 222 times faster than MINT).

In terms of threats to validity, the size of the log files is a confounding factor that could affect our results (i.e., accuracy and execution time). We mitigated such a threat by considering seven datasets with different sizes (ranging from 5K to 35K log entries) and different sets of system executions.

4.5 Conclusion

In this chapter, we addressed the scalability problem of inferring the model of a component-based system from the individual component-level logs, assuming only limited (and possibly incomplete) knowledge about the system. Our approach, called SCALER, first infers a model of each system component from the corresponding logs; then, it merges the individual component models together taking into account the dependencies among components, as reflected in the logs. Our evaluation, performed on logs from an industrial system, has shown that SCALER can process larger logs, is faster, and yields more accurate models than a state-of-the-art technique.

Chapter 5

Test Case Decomposition

Regression testing is arguably one of the most important activities in software testing. However, its cost-effectiveness and usefulness can be largely impaired by complex system test cases that are poorly designed (e.g., test cases containing multiple test scenarios combined into a single test case) and that require a large amount of time and resources to run. One way to mitigate this issue is decomposing such system test cases into smaller, separate test cases—each of them with only one test scenario and with its corresponding assertions—so that the execution time of the decomposed test cases is lower than the original test cases, while the test effectiveness of the original test cases is preserved. This decomposition can be achieved with program slicing techniques, since test cases are software programs too. However, existing static and dynamic slicing techniques exhibit limitations when (1) the test cases use external resources, (2) code instrumentation is not a viable option, and (3) test execution is expensive.

In this chapter, we propose a novel approach, called DS3 (Decomposing System teSt caSe), which automatically decomposes a complex system test case into separate test case slices. The idea is to use test case execution logs, obtained from past regression testing sessions, to identify "hidden" dependencies in the slices generated by static slicing. Since logs include run-time information about the system under test, we can use them to extract access and usage of global resources and refine the slices generated by static slicing.

The rest of the chapter is organized as follows. Section 5.1 provides the context. Section 5.2 illustrates the motivating example. Section 5.3 describes the main steps of DS3. Section 5.4 reports on the evaluation of DS3. Section 5.5 discusses the practical implications of using DS3. Section 5.6 concludes this chapter.

5.1 Overview

Regression testing is a quality assurance technique applied when changes are made to an existing codebase. It provides confidence that the performed changes do not harm the behavior of the existing and unchanged parts of the code [133]. Although many techniques have been introduced for cost-effective regression testing, such as test case prioritization and test suite selection, their usefulness can be largely impaired if individual test cases (1) are poorly designed (e.g., the codebase contains test smells) and (2) require a large amount of time and resources to run [7, 109].

We observed both phenomena in the context of a collaborative industrial research project, with a large company in the aerospace domain. Due to the intrinsic complexity of the system under test (SUT) and to the accumulated technical debt over several years of software development, system test cases often contain multiple test scenarios combined into a single test case. These tests, often called *eager tests* [119], negatively impact both regression testing and test evolution. In our industrial context, *eager tests* are very expensive as they take several hours to run. This means that, for example, even if a state-of-the-art test case prioritization technique is applied, no faults could be detected during the first few hours of test execution. Furthermore, eager tests are more difficult to read, understand, document, and evolve [119].

However, if such complex system test cases could be decomposed into smaller test cases without losing their test effectiveness, the execution time of the decomposed test cases would decrease and the cost-effectiveness of test case prioritization could improve. Moreover, having smaller test cases would facilitate fault localization [131] and program maintenance [7, 119] by providing more granular information about test results. Furthermore, as decomposed test cases have distinct test assertions, engineers would be able to easily pick specific test cases of interest and run them efficiently.

To achieve this, ideally, one would decompose a complex system test case containing multiple test scenarios into separate system test cases, each of them with only one test scenario and its corresponding assertions. Since system test cases are software programs too, they could be decomposed using static slicing techniques based on def-use analysis [126]. However, existing *static slicing* techniques cannot identify and deal with “hidden” dependencies between statements, originated from the usage of global resources such as external files or databases; such missing dependencies would lead to run-time errors in the resulting decomposed (sliced) test cases. *Dynamic slicing* [14] could be an alternative to static slicing, but it requires to instrument the source code and to collect coverage information. However, this alternative is not feasible for a system composed of third-party components and it would not address the problem of handling “hidden” dependencies as they are not captured by code coverage. Finally, *observational slicing* [14, 132] would require running each system test case multiple times. Such an approach is not applicable for typically long system test case execution times, as it is the case for the system developed by our industrial partner.

In this chapter, we tackle the problem of slicing a complex system test case into simpler ones — without missing any “hidden” dependency between statements — by proposing a novel approach, called *DS3 (Decomposing System teSt caSe)*, which complements static slicing with a log-based analysis. The idea is to use test case execution logs, obtained from past regression testing sessions, to identify missing dependencies in the decomposed test cases generated by static slicing. Since logs include run-time information about the SUT, we can use them to extract the global resources accessed (e.g., files, databases) and the actions performed (e.g., read/write file, open/close database) upon executing each statement in the original (unsliced) system test case. In this way, we can reconstruct the additional dependencies between statements as defined by the usage of global resources.

DS3 first generates test slices (i.e., decomposed test cases) by applying backward static slicing using individual assertions included in the original system test case as slicing criteria. Then, DS3 complements the individual slices taking into account any missing dependencies identified by the analysis of the test execution logs.

We implemented DS3 in a prototype tool, on top of an off-the-shelf program slicing tool. We evaluated DS3 in terms of slicing effectiveness (i.e., ability to identify all required dependencies) and compared it with the vanilla static slicing tool. We also compared the test case slices obtained with DS3 with the corresponding unsliced system test case, in terms of efficiency and effectiveness, i.e., how quickly we can verify individual assertions and how many faults we can detect. In our evaluation, we used one proprietary system provided by our industrial partner and one open-source system. The results show that DS3 is able to accurately identify the dependencies related to the usage of global resources, which vanilla static slicing misses. Moreover, decomposed test cases are much faster than the corresponding original system test case (with an average speedup of 3.56x) and there is no significant loss in terms of effectiveness (fault detection rate). Additionally, both the original system test cases and the decomposed test cases have the same function coverage with a small difference in branch coverage.

To summarize, the main contributions of this chapter are:

1. DS3, an approach for slicing complex system test cases using the global resources usage information available in test case execution logs;
2. the evaluation of DS3 in terms of slicing effectiveness as well as test efficiency and effectiveness.

5.2 Motivating Example

In this section, we present an example that motivates our log-based slicing idea. Figure 5.1 shows, on the top, an example system test case $tc_{sys} = \langle s_1, s_2, \ldots, s_7 \rangle$ (where s_i is the i-th statement). Figure 5.2 shows tc_{sys}'s corresponding execution log $l_{sys} = \langle e_1, \ldots, e_4 \rangle$; for simplicity, we show the structured log instead of its original, free-formed log and omit log entries not related to the usage of global resources. Notice that each log entry has a reference to the test case statement originating it. This example is a simplified version of a system test case in *JSBSim* [58], an open-source flight simulator.

The example test case has been designed to cover multiple test scenarios, with the presence of two assertions (i.e., s_4 and s_7). Such test cases can become less than an ideal if an engineer is interested in testing only a specific scenario, and the execution of *both* test scenarios is expensive.

Ideally, tc_{sys} could be replaced with two test cases $I_1 = \langle s_1, s_2, s_3, s_4 \rangle$ and $I_2 = \langle s_1, s_2, s_5, s_6, s_7 \rangle$, as shown at the middle and at the bottom of Figure 5.1, each of which contains only an assertion referring to a specific test scenario. In this way, an engineer can select which test scenario to execute, reducing the overall testing cost. Notice that, though the new test cases are smaller than the original system test case, executing I_1 and I_2 is equivalent to executing tc in terms of code coverage.

An engineer may try to use program slicing to generate I_1 and I_2 from tc_{sys} since tc_{sys} is a software program too and I_1 and I_2 can be seen as slices of tc_{sys}. In particular, to have one assertion per slice, the engineer could apply backward static slicing using $\langle s_4, \{\texttt{ref}, \texttt{sim}\} \rangle$ and $\langle s_7, \{\texttt{diff}\} \rangle$ as slicing criteria. However, the variable `fdm` defined in s_1 is never used in the following statements in tc_{sys}, and therefore none of the slices generated based on the data- and control-flow of the program code contains s_1. This is critical because,

```
def test_example():
    (1) fdm = create_fdm_setup()
    (2) ref = read_csv('output.csv')
    (3) sim = deploy_proc('output.csv')
    (4) self.assertEqual(ref, sim)
    (5) new = run_ic()
    (6) diff = FindDiffs(ref, new, 1E-8)
    (7) self.assertEqual(len(diff), 0)

def test_example_ideal_slice1():
    (1) fdm = create_fdm_setup()
    (2) ref = read_csv('output.csv')
    (3) sim = deploy_proc('output.csv')
    (4) self.assertEqual(ref, sim)

def test_example_ideal_slice2():
    (1) fdm = create_fdm_setup()
    (2) ref = read_csv('output.csv')
    (5) new = run_ic()
    (6) diff = FindDiffs(ref, new, 1E-8)
    (7) self.assertEqual(len(diff), 0)
```

Figure 5.1: A system test case tc_{sys} (top) and its ideal slices I_1 (middle) and I_2 (bottom)

as recorded in l_{sys}, file `output.csv` needed in s_2 is internally generated by `create_fdm_setup()` in s_1. In practice, this means that the execution of the sliced test cases generated with vanilla static slicing will result in a crash, due to the missing resource (file `output.csv`).

Overall, because of the "hidden" dependency between s_1 and s_2, static slicing alone cannot properly generate I_1 and I_2 from tc_{sys}.

This simple example has shown the need for extending static slicing to identify hidden dependencies due to the usage of global resources. In the next section, we will present a method that achieves this goal leveraging the information contained in test case execution logs.

ID	Statement	Template	Value
e_1	s_1	`read file *`	`setup.xml`
e_2	s_1	`write file *`	`output.csv`
e_3	s_2	`read file *`	`output.csv`
e_4	s_3	`read file *`	`output.csv`

Figure 5.2: The execution log l_{sys} for tc_{sys}

5.3 Log-based System Test Case Decomposition

Our new approach, called DS3, decomposes a complex system test case containing multiple test scenarios into multiple individual system test cases, each of them with only one test scenario and its related subset of assertions, while preserving the hidden dependencies due to the usage of global resources. The main idea is to complement static slicing with a log-based analysis. Since test execution logs include run-time information about the system under test, we can use them to extract the global resources accessed (e.g., files, databases) and the actions performed (e.g., read, write) upon executing each statement in the original (unsliced) system test case. In this way, we can reconstruct *hidden* dependencies between statements generated at run-time by global resources, which were not identified by static slicing.

DS3 takes as input a system test case, an execution log corresponding to the test case, and the log message templates related to global resources; it returns a set of slices, each of them exercising an individual test scenario and containing fewer assertions. In our running example, DS3 takes the system test case tc_{sys} (Figure 5.1, top) and its corresponding log l_{sys} (Figure 5.2) and returns ideal slices I_1 and I_2 (Figure 5.1, middle and bottom). The engineer is only required to mark log message templates related to global resources, such as `output.csv`, in the log. For example, the log entry e_1 in l_{sys} indicates that the "read" operation is performed on file `setup.csv`. Hence, by looking at each message template, such as `read file *`, engineers can easily identify if it is related to the usage of global resources. Then, DS3 automatically identifies the hidden dependency between s_1 and s_2 using a log-based analysis, and generates I_1 and I_2 by refining the intermediate slices generated by static slicing.

Note that our approach is black-box: it does not require access to the source code. Therefore, it can be applied to software systems composed of

3rd-party components, whose source code is not accessible, as it is the case for the system developed by our industry partner. Nevertheless, we need two conditions to be satisfied to apply DS3: 1. there is a traceability information between statements in the test case and messages in the log and 2. the log contains some information on the usage of global resources. These conditions are required to identify 1) which messages were logged upon the execution of each statement of the test case and 2) the global resources used as part of the statement execution. Such conditions are satisfied by the system developed by our industry partner. In general, such conditions can easily be satisfied by appropriately instrumenting test cases and adding a watchdog process (with logging capabilities) to monitor the usage of global resources at run time. Though DS3 additionally requires engineers to manually mark message templates related to global resources, the number of all templates is typically manageable (e.g., there are 14 message templates in our proprietary system), and it is easy for engineers with domain knowledge to identify the templates related to the usage of global resources.

Algorithm 6 provides the pseudo-code of DS3. It takes as input a system test case $tc = \langle s_1, \dots, s_n \rangle$, its corresponding execution log $l = \langle e_1, \dots, e_k \rangle$, and the set of log messages templates $MT_G = \{et_1, \dots, et_m\}$ marked as related to the usage of global resources in l; it returns a set of decomposed test cases (i.e., slices) $\mathcal{D} = \{D_1, \dots, D_j\}$.

Algorithm 6 consists of four major stages: 1. assertion-based backward slicing (lines 1–6), 2. def-use analysis for global resources using logs (line 7), 3. slice refinement (lines 8–14), and 4. slice minimization (line 15). The backward slicing stage generates static slices $\mathcal{D}$ from tc. The global resources def-use analysis stage identifies the relationships between the statements in tc, the log entries in l related to global resources, and the actions performed on the latter, using l and MT_G. The resulting set of triples G_{du} is then used to refine each of the static slices $D \in \mathcal{D}$ in the slice refinement stage. Last, the slice minimization stage removes any redundant slices in $\mathcal{D}$. The algorithm ends by returning the minimized $\mathcal{D}$. The four stages are described in detail in the following subsections.

5.3.1 Assertion-based Backward Slicing

Algorithm 6 first performs backward static slicing on the assertions in tc to generate a set of static slices $\mathcal{D}$ (lines 1–6). This guarantees that each $D \in \mathcal{D}$ has at most one test scenario by having one assertion. The algorithm starts by initializing $\mathcal{D}$ as an empty set (line 1). For each assertion statement $s \in tc$ (lines 2–6), the algorithm gets the variables V_s in s (line 3), performs the

Algorithm 6 DS3: Decomposing System Test Case

```
Input: System Test Case $tc = \langle s_i, \dots, s_n \rangle$
       Log $l = \langle e_1, \dots, e_k \rangle$
       Set of Templates $MT_G = \{et_1, \dots, et_m\}$
Output: Set of Decomposed Test Cases $\mathcal{D} = \{D_1, \dots, D_j\}$
 1: Set of Test Cases $\mathcal{D} \leftarrow \emptyset$
 2: for Assertion Statement $s \in tc$ do
 3:     Set of Variables $V_s \leftarrow$ GET-VARS($s$)
 4:     Test Case $D_s \leftarrow$ BACK-SLICE($s, V_s, tc$)
 5:     $\mathcal{D} \leftarrow \mathcal{D} \cup \{D_s\}$
 6: end for
 7: Set of Triples $G_{du} \leftarrow$ GLOBAL-DU($l, tc, MT_G$)
 8: for Decomposed Test Case $D \in \mathcal{D}$ do
 9:     Test Case $D_{tmp} \leftarrow D$
10:     for Statement $s \in D_{tmp}$ do
11:         Set of Statements $W \leftarrow$ DEP-STMTS($s, G_{du}, tc$)
12:         $D \leftarrow$ ADD($D, W$)
13:     end for
14: end for
15: return MINIMIZE($\mathcal{D}$)
```

backward static slicing on tc using $\langle s, V_s \rangle$ as the slicing criterion (line 4), and adds the resulting slice D into $\mathcal{D}$ (line 5).

In our running example tc_{sys}, for the assertion $s_4 \in tc_{sys}$, the algorithm performs the backward static slicing using $\langle s_4, \{\texttt{ref}, \texttt{sim}\} \rangle$ as the slicing criterion, yielding the slice $D_{s_4} = tc_{static_1} = \langle s_2, s_3, s_4 \rangle$, shown at the top of Figure 5.3. Note that s_1 is not included in tc_{static_1} because the variable `fdm` is not used in any statements in tc_{static_1} based on the static def-use analysis. Similarly, for the second assertion $s_7 \in tc_{sys}$, the backward slicing with the slicing criterion $\langle s_7, \{\texttt{diff}\} \rangle$ yields another slice $D_{s_7} = tc_{static_2} = \langle s_2, s_5, s_6, s_7 \rangle$, shown at the bottom of Figure 5.3. tc_{static_2} does not include s_1, s_3, and s_4 as they do not affect the computation of the statements in tc_{static_2} based on the static analysis.

5.3.2 Def-Use Analysis for Global Resources

The second stage of Algorithm 6 identifies a set of triples G_{du} in tc using l and MT_G (line 7) where each triple $\langle s, g, a \rangle \in G_{du}$ indicates that an action a is performed on a global resource g when the statement $s \in tc$ is executed.

```
def test_example_static_slice1():
  (2) ref = read_csv('output.csv')
  (3) sim = deploy_proc('output.csv')
  (4) self.assertEqual(ref, sim)

def test_example_static_slice2():
  (2) ref = read_csv('output.csv')
  (5) new = run_ic()
  (6) diff = FindDiffs(ref, new, 1E-8)
  (7) self.assertEqual(len(diff), 0)
```

Figure 5.3: Static slice results: tc_{static_1} (top) and tc_{static_2} (bottom)

This is done by Algorithm 7.

Algorithm 7 GLOBAL-DU

```
Input: Log $l = \langle e_1, \ldots, e_k \rangle$
       System Test Case $tc = \langle s_1, \ldots, s_n \rangle$
       Set of Templates $MT_G = \{et_1, \ldots, et_m\}$
Output: Set of Triples $G_{du}$
 1: $G_{du} \leftarrow \emptyset$
 2: for Statement $s \in tc$ do
 3:     Set of Log Entries $E_s \leftarrow$ ENTRY-FOR-STATEMENT($s, l$)
 4:     for Log Entry $e \in E_s$ do
 5:         if TEMPLATE($e$) $\in MT_G$ then
 6:             String $g \leftarrow$ GET-GLOBAL-RESOURCE($e$)
 7:             Action $a \leftarrow$ GET-ACTION-TYPE($e$)
 8:             $G_{du} \leftarrow G_{du} \cup \{\langle s, g, a \rangle\}$
 9:         end if
10:     end for
11: end for
12: return $G_{du}$
```

Algorithm 7 starts by initializing G_{du} as an empty set. For each statement $s \in tc$ (lines 2–11), the algorithm identifies the set of log entries E_s originated from s using the traceability information between the statements in tc and the log entries in l (line 3). Then, for each log entry $e \in E_s$ (lines 4–10) whose template is in MT_G (line 5), the algorithm identifies the global resource g from e's parameter value (line 6) and the action type a from e's template (line 7). If the template contains predefined keywords indicating a "use" of the resource, such as `read` and `access`, then $a = \texttt{use}$; similarly, keywords such as `write` and `update` indicate a "definition" of a resource and we have $a = \texttt{def}$. The default set of keywords characterizing uses and definitions of global resources can be enhanced by engineers. The identified triple $\langle s, g, a\rangle$ is added into G_{du} (line 8); the algorithm ends by returning G_{du}.

In our running example, let us consider the case where GLOBAL-DU is called with parameters $tc = tc_{sys} = \langle s_1, \dots, s_7\rangle$, $l = l_{sys} = \langle e_1, \dots, e_4\rangle$, and $MT_G = \{\texttt{read file *}, \texttt{write file *}\}$. The algorithm first initializes G_{du} to $\emptyset$ and starts the iteration over statements in tc_{sys}. For $s_1 \in tc_{sys}$, the call to ENTRY-FOR-STATEMENT with s_1 and l_{sys} returns $E_s = \{e_1, e_2\}$, by checking the reference of each log entry to the test case statement originating in l_{sys}. Based on E_s, the algorithm starts the inner iteration over entries in E_s. For $e_1 \in E_s$, since e_1's template, i.e., `read file *`, is in MT_G, the algorithm identifies the global resource and action type of e_1 using GET-GLOBAL-RESOURCE and GET-ACTION-TYPE. Specifically, the call to GET-GLOBAL-RESOURCE with e_1 checks the parameter value of e_1 and returns $g = \texttt{setup.xml}$. Similarly, the call to GET-ACTION-TYPE with e_1 checks the template of e_1 and returns $a = \texttt{use}$ because `read file *` contains the `read` keyword. The algorithm ends the inner iteration for e_1 by adding the triple $\langle s_1, \texttt{setup.xml}, \texttt{use}\rangle$ into G_{du} and moves on to the next iteration to process e_2. After processing all statements and corresponding log entries, the algorithm returns $G_{du} = \{\ \langle s_1, \texttt{setup.xml}, \texttt{use}\rangle, \langle s_1, \texttt{output.csv}, \texttt{def}\rangle, \langle s_2, \texttt{output.csv}, \texttt{use}\rangle, \langle s_3, \texttt{output.csv}, \texttt{use}\rangle\ \}$.

5.3.3 Log-based Slice Refinement

The two previous stages of Algorithm 6 compute $\mathcal{D}$ and G_{du}. In this stage, we aim to refine the slices $\mathcal{D}$ using the information in G_{du} (lines 8–14). Specifically, for each slice $D \in \mathcal{D}$ and for each statement $s \in D$, the algorithm finds all the statements of tc needed for s using algorithm DEP-STMTS (line 11, described in detail below) and adds the found statements into D (line 12). As a result, we ensure that all $D \in \mathcal{D}$ have no missing statements, and can be executed without resulting in a crash due to the improper usage of a global resource (e.g., writing to a file before opening it).

Algorithm 8 presents the pseudo-code of DEP-STMTS, the core of the slice refinement stage. It *recursively* finds dependent statements using both the information in G_{du} and backward static slicing[1].

[1]Note that Algorithm 8 could be called for the same statement multiple times. To reduce

Algorithm 8 DEP-STMTS

```
Input: Statement $s$
       Set of (Global Resource def-use) Triples $G_{du}$
       System Test Case $tc = \{s_1, \ldots, s_n\}$
Output: Set of Statements $S$
 1: Set of Variables $V_s \leftarrow$ GET-VARS($s$)
 2: Set of Statements $S_{dir} \leftarrow$ DEFS($s, G_{du}$) $\cup$ BACK-SLICE($s, V_s, tc$)
 3: if $S_{dir} = \emptyset$ then
 4:     return $\emptyset$
 5: else
 6:     Set of Statements $S_{rec} \leftarrow \emptyset$
 7:     for Statement $s_{dir} \in S_{dir}$ do
 8:         $S_{rec} \leftarrow S_{rec} \cup$ DEP-STMTS($s_{dir}, G_{du}, tc$)
 9:     end for
10:     return $S_{dir} \cup S_{rec}$
11: end if
```

For a given s, G_{du}, and tc, the algorithm first gets a set of variables V_s in s and calculates a set of statements S_{dir} directly needed for s using DEFS and BACK-SLICE. Specifically, the call to DEFS with s and G_{du} returns all statements s' such that $\langle s', g, \texttt{def}\rangle \in G_{du}$ and $\langle s, g, \texttt{use}\rangle \in G_{du}$. The call to BACK-SLICE with s, V_s, and tc returns all statements in the static slice of tc constructed by the slicing criterion of $\langle s, V_s\rangle$. If $S_{dir} = \emptyset$, the algorithm returns $\emptyset$ (line 4); otherwise, the algorithm collects another set of statements S_{rec} needed for each statement $s_{dir} \in S_{dir}$ (lines 6–9) by recursively calling DEP-STMTS with s_{dir} (line 8) and then returns $S_{dir} \cup S_{rec}$ (line 10).

In our running example, let us consider the case where DEP-STMTS is called with parameters $s = s_2$, $G_{du} = \{\langle s_1, \texttt{setup.xml}, \texttt{use}\rangle, \langle s_1, \texttt{output.csv}, \texttt{def}\rangle, \langle s_2, \texttt{output.csv}, \texttt{use}\rangle, \langle s_3, \texttt{output.csv}, \texttt{use}\rangle\}$, and $tc = tc_{sys}$. Since s_2 has no variables, $V_s = \emptyset$ and the call to BACK-SLICE returns $\emptyset$. On the other hand, the call to DEFS returns $\{s_1\}$ since $\langle s_1, \texttt{output.csv}, \texttt{def}\rangle \in G_{du}$ and $\langle s_2, \texttt{output.csv}, \texttt{use}\rangle \in G_{du}$. Thus, $S_{dir} = \{s_1\}$, and the algorithm recursively calls DEP-STMTS for s_1. Since s_1 does not depend on any other statement, the recursive call returns $\emptyset$, leading to $S_{rec} = \emptyset$. The algorithm ends by returning $S_{dir} \cup S_{rec} = \{s_1\}$.

Recall that the assertion-based backward slicing stage calculated $D_{s_4} = tc_{static_1} = \langle s_2, s_3, s_4\rangle$ and $D_{s_7} = tc_{static_2} = \langle s_2, s_5, s_6, s_7\rangle$ for tc_{sys}. Since the call

the execution time, Algorithm 8 internally keeps a cache of the dependency information for each statement.

to DEP-STMTS for s_2 returns $\{s_1\}$ as described above, both D_{s_4} and D_{s_7} can be refined, thanks to the inclusion of s_1. Note that, while s_3 also depends on s_1 according to G_{du}, function ADD in Algorithm 6 (line 12) does not redundantly add s_1 to D_{s_4} and D_{s_7}. As a result, D_{s_4} and D_{s_7} become the same as the ideal slices I_1 and I_2, respectively.

5.3.4 Slice Minimization

After the slice refinement stage, $\mathcal{D}$ may contain slices that (excluding the assertions) are subsets of others. A slice D_i is a *subset* of another slice D_j, denoted by $D_i \sqsubseteq D_j$, if all statements (except assertions) of D_i are in D_j. As we are dealing with test case slices, we make the following assumption, based on our observations in real-word codebases: $D_i \sqsubseteq D_j$ (or $D_j \sqsubseteq D_i$) holds when D_i and D_j belong to the same test scenario and thus share the same test fixture (e.g., the same setup and teardown code). For example, a test scenario for the initialization routine of an object can be implemented with multiple assertions that verify the initialization of the various properties of the object; in such a case, the assertions will rely on the same setup code. Based on this assumption, DS3 includes a minimization stage to remove any redundant slices in terms of test scenarios. Specifically, the MINIMIZE function in Algorithm 6 analyzes each obtained slice (ignoring assertions) in $\mathcal{D}$, to ensure that the property $\forall D_i, D_j \in \mathcal{D},\ \ D_i \not\sqsubseteq D_j \wedge D_j \not\sqsubseteq D_i$ holds on $\mathcal{D}$.

If MINIMIZE finds two slices $D_i, D_j \in \mathcal{D}$ such that $D_i \sqsubseteq D_j$, it merges D_i and D_j by moving the assertion from D_i into D_j (preserving the order among statements defined in the original test case) and removing D_i from $\mathcal{D}$.

In our running example, when ignoring the assertion statements, $D_{s_4} = I_1$ contains s_3 and s_4 that are not included in $D_{s_7} = I_2$, and therefore $D_{s_4} \not\sqsubseteq D_{s_7}$; similarly, $D_{s_7} \not\sqsubseteq D_{s_4}$. So MINIMIZE returns $\{D_{s_4}, D_{s_7}\} = \{I_1, I_2\}$, and Algorithm 6 ends by returning $\mathcal{D} = \{I_1, I_2\}$.

Note that the minimization stage results in some slices having multiple assertions since MINIMIZE merges slices without taking them into account. Nevertheless, based on the aforementioned assumption, the assertions included in each slice belong to the same test scenario. Furthermore, the assertions of the original test case will be distributed over the generated slices, thus reducing the overall number of assertions per slice.

5.4 Evaluation

We implemented DS3 as a `Python` program, using the `Python-Program-Analysis` toolkit [99] to perform static slicing.

In this section, we report on the assessment of DS3 in slicing system test cases, and the *effectiveness* and *efficiency* of the obtained slices. More specifically, our research is steered by the following research questions:

RQ1: *How effective is DS3 in slicing system test cases compared to standard static slicing?*

RQ2: *How efficient are the slices produced by DS3 compared to the original test cases?*

RQ3: *What is the code coverage and fault detection capability of the slices produced by DS3 compared to the original test cases?*

RQ1 investigates how effective DS3 is in slicing system test cases into sliced test cases that successfully compile and have no run-time errors (i.e., no "hidden" dependency is missing). These two aspects are essential when decomposing complex system test cases since slices yielding compilation or run-time errors are useless.

RQ2 assesses the running time (efficiency) of the generated test slices compared to the corresponding original (non-sliced) test cases. Efficiency is important in the context of regression testing because developers would execute only a subset of the test cases (and their assertions) to find regression faults within the available time budget. The efficiency of regression testing depends on the running time of the test cases that are selected.

RQ3 analyzes the code coverage and fault detection capability of the generated test slices compared to the non-sliced ones. Since DS3 decomposes system test cases into slices, a potential drawback is that the latter may be less effective than the former in terms of structural coverage and fault detection capability. Thus, it is essential to investigate how structural coverage and fault detection capability may be affected by applying DS3.

5.4.1 Benchmarks

A candidate benchmark for our evaluation should meet the following requirements: (1) it contains system or integration level test cases, (2) the test cases should generate logs when executed, and (3) the test cases should access/use global resources (e.g., external files, databases, remote resource).

These requirements are fulfilled by a proprietary benchmark, hereafter referred to as *Prop*, provided by one of our industrial partners active in the satellite industry. This benchmark includes 30 complex system test cases written in Python, each of which takes on average 53 minutes to execute, as it triggers multiple cyber-physical components[2].

[2]Due to non-disclosure agreements, we cannot divulge more details about this system.

To increase the diversity of our experimental subjects and support open science, we aimed to include in our benchmark one open-source system as well. Among the top 10 trending repositories on GitHub, we filtered out those that do not satisfy the above requirements, ending up with one open-source system, namely *JSBSim*, an open-source flight simulator already introduced in our running example in section 5.2. It is mainly written in C++ with about 300 source files (in total over 20 KLOC). It also includes 81 system-level test cases written in Python.

Our approach uses logs to detect dependencies originated from the usages of global resources. The proprietary test cases generate log messages with detailed information regarding the usage of global resources (file, database, and network connection) and the timestamp of each executed test statement. Therefore, *Prop* did not require further modifications to generate appropriate logs.

In contrast, the logs generated by the *JSBSim* test cases do not include the usages of global resources by default. Thus, we implemented a *watchdog* script that monitors the usage of global resources during test executions. Since the system test cases in *JSBSim* access and modify external files only, our script captured the names of the changed files as well as the timestamp of each file access. Besides, we also instrumented the test cases to store the timestamps of each executed test statement. This allowed us to determine precisely which statement read or wrote which external files. Notice that implementing a watchdog and instrumenting system test cases can easily be automated, without requiring access to the code base.

5.4.2 RQ1: Slicing Effectiveness

5.4.2.1 Methodology

To answer RQ1, we considered static slicing (using the same slicing criterion used in the assertion-based backward slicing stage of DS3, see 5.3.1) as the baseline for comparison; we used the implementation provided by the `Python-Program-Analysis` toolkit, since all the test cases in our benchmarks are written in Python. We consider neither dynamic slicing nor observational slicing as alternative baselines. The former is not feasible as our proprietary system *Prop* includes several third-party components; hence, instrumenting these components and building the test execution traces is not possible. The latter is too expensive for system test cases since it requires executing each test case multiple times, each time by deleting one single test statement and observing whether the test case fails [14]. Considering the average execution time of 53 minutes of the system test cases in the *Prop* benchmark, observational slicing was not applicable from a practical standpoint.

To assess the slicing effectiveness, we ran both DS3 and the baseline static slicing tool on the two benchmarks, obtaining two sets of sliced test cases (one generated by each tool). To measure the slicing effectiveness of the two approaches, we used the following metric:

$$Eff(\mathcal{D}) = \frac{|\{\mathcal{D}_i \in \mathcal{D} \mid \mathcal{D}_i \text{ has no errors}\}|}{|\mathcal{D}|}$$

where $\mathcal{D}$ is the set of slices produced by a given approach (DS3 or the baseline). In the formula, the numerator indicates the number of test slices that do not lead to compilation or run-time errors; the denominator represents the total number of generated slices. The value of *Eff* ranges between 0 and 1; larger values are preferable as they indicate fewer failing test slices. In our context, generated slices may fail due to missing dependencies; therefore, larger $Eff(\mathcal{D})$ values mean that the technique under analysis is more effective in generating correct slices that do not miss any dependency.

Notice that the same static slicer used in the first stage of DS3 is also used as baseline static slicer. This means that the comparison between DS3 and the baseline actually shows the effect of log-based refinement on vanilla static slicing.

5.4.2.2 Results

Table 5.1 reports the number of slices produced by both our approach and the baseline. Column "System" indicates the name of the benchmark; column "# Test Cases" indicates the number of the original test cases to be sliced; columns "Total", "Pass", and "Fail" indicate, respectively, the total number of obtained slices, the number of test case slices that successfully passed when executed, and the number of test case slices that failed when executed; column "$Eff(\mathcal{D})$" indicates the slicing effectiveness of an approach. We remark that the total number of test cases to be sliced for *JSBSim* is 76 because we could not get the results for five out of 81 test cases: three of them were flaky (i.e., passing in some runs and failing in others) and two further test cases could not be properly parsed by the static slicer.

The results show that the static slicer has lower slicing effectiveness than DS3 for both subject systems. For *JSBSim*, the static slicer achieved an effectiveness score of 0.33. Indeed, 67% of the test slices it generated resulted in compilation or run-time errors; such errors occurred because the static slicer missed hidden dependencies. For *Prop*, only 24% of the slices generated by the

Table 5.1: Comparison between DS3 and static slicing in terms of slicing effectiveness $\mathit{Eff}(\mathcal{D})$

System	# Test Cases	Approach	#Slices			$\mathit{Eff}(\mathcal{D})$
			Total	#Pass	#Fail	
JSBSim	76	MoLFI	84	84	0	1
		Static Slicer	169	56	113	0.33
Prop	30	MoLFI	137	137	0	1
		Static Slicer	166	40	126	0.24

static slicer ran successfully, without run-time errors, as they had no missing dependencies. Instead, all the slices generated by DS3 ran successfully, without errors, resulting in an effectiveness score of 1. Overall, this means that DS3, leveraging the global resources usages recorded in the logs, is able to identify many hidden dependencies that a vanilla static slicer would have missed.

5.4.3 RQ2: Efficiency of the Sliced Test Cases

5.4.3.1 Methodology

To answer RQ2, we compared the execution time of the generated test slices with the execution time of the original (non-sliced) test cases. As the test case execution can alter the environment (e.g., by creating or modifying a file), we reset the environment before running each test case, to avoid any incorrect results. We ran each test 10 times to account for the uncertainty in test execution time. We also assessed the overhead of DS3 by measuring, over the 10 executions, the average time taken by DS3 internal stages (see Algorithm 6 in section 5.3) and the total execution time for slicing a given test case.

The *JSBSim* test cases have a very short execution time (a few seconds), so they are not adequate to realistically assess the efficiency of the sliced test cases. For this reason, to answer RQ2, we only considered the results obtained for the *Prop* test cases.

All test cases (original and sliced) were executed on an Apple MacBook Pro computer with a 2.5GHz Intel Core i7 processor and 16G of memory.

5.4.3.2 Results

Table 5.2 shows the time (in seconds) for running DS3 to generate the test slices, as well as the time for executing the generated slices and the original test cases. More specifically, columns “Others”, “Static”, and “Refine” indicate the average time for executing the different stages of DS3: finding global resources defs and uses and the minimization step (column “Others”), static slicing (column “Static”), slice refinement (column “Refine”); column “Total” indicates the average total execution time of DS3; column “Slices” indicates the number of slices produced by DS3; column “Org” indicates the execution time of the original (non-sliced) test cases; columns “Sl. Avg” and “Sl. Tot” indicate the average and the cumulative execution time of the sliced test cases, respectively, where the latter represents the sum of the execution time of all slices obtained for each individual system test case; column “Speedup” indicates the speedup ratio between the execution time of the non-sliced test cases and the cumulative execution time of the sliced test cases.

The results shows that, for 24 out of 30 test cases, DS3 produced test case slices whose cumulative execution time is shorter than the one of the original test case, with an average speedup of 3.56x.

The largest speedup (23.14x) can be observed for test case PTC30. In this case, the cumulative execution time of the five slices is 1004 s ($\approx$ 17 minutes) on average; instead, executing the original test case requires 23 231 s ($\approx$ 387 minutes, i.e., more than six hours). This large difference is due to the execution, within the original test case, of an expensive procedure that actually does not have any dependencies with other statements in the test case; indeed, DS3 successfully determined and excluded this procedure in the test case slices it generated.

In the remaining six test cases (characterized by a speedup ratio lower than one), the total execution time of the test slices was higher than the one of the original test case. We observed the lowest value of the speedup ratio (i.e., the highest slowdown, 0.53) for PTC5: the execution time of the original test case was 465 s while the cumulative execution time of the two slices produced by DS3 was 874 s. To further understand the root cause of this large increase in execution time, we manually analyzed PTC5 and its corresponding slices. We discovered that, for this case, DS3 created two independent test slices, each with the same copy of the test set-up code; executing this set-up code takes a large portion of the execution time of the text case slice.

Statistical Analysis We further analyzed the results reported in Table 5.2 using statistical and effect size tests. In particular, we used the Wilcoxon

Table 5.2: Execution time (in seconds) of DS3 and of the original and sliced system test cases.

STC	DS3 (s)				Slices	Test Cases (s)			Speedup
	Others	Static	Refine	Total		Org	Sl. Avg	Sl. Tot	
PTC1	2.76	5.06	204.42	212.23	2	1139	520.00	1040	1.10
PTC2	2.19	3.25	108.79	114.21	2	498	342.50	685	0.73
PTC3	0.14	3.08	32.99	36.15	3	245	132.60	398	0.62
PTC4	3.51	3.07	54.11	60.70	1	330	210.00	210	1.57
PTC5	0.10	3.57	84.65	88.35	2	465	437.00	874	0.53
PTC6	2.77	3.07	36.31	42.25	2	2720	139.00	278	9.78
PTC7	13.84	18.53	258.58	290.94	3	3080	372.00	1116	2.76
PTC8	18.78	25.36	260.06	304.18	4	3765	404.75	1619	2.33
PTC9	0.75	5.48	15.49	21.72	1	197	103.00	103	1.91
PTC10	10.24	68.15	89.47	167.62	7	2280	296.43	2075	1.10
PTC11	9.50	4.95	20.13	34.58	3	1041	121.00	363	2.87
PTC12	4.26	9.95	69.89	84.10	3	913	182.67	548	1.67
PTC13	3.65	50.81	28.29	82.74	5	4150	328.60	1643	2.53
PTC14	0.01	27.81	91.25	119.07	7	15480	144.00	1008	15.36
PTC15	4.16	5.66	9.50	19.32	2	541	89.00	178	3.04
PTC16	0.51	49.58	142.25	192.32	6	922	128.50	771	1.20
PTC17	5.08	11.43	596.04	612.53	8	4975	212.50	1700	2.93
PTC18	0.04	11.46	11.14	22.61	5	4091	116.00	580	7.05
PTC19	0.02	5.08	36.11	41.19	3	183	68.33	205	0.89
PTC20	0.08	15.10	223.85	239.02	3	342	70.00	210	1.63
PTC21	0.06	20.79	136.81	157.65	2	4172	953.50	1907	2.19
PTC22	0.03	9.84	27.52	37.37	5	1201	209.00	1045	1.15
PTC23	0.11	15.95	250.19	266.24	3	6383	281.67	845	7.55
PTC24	0.03	9.24	31.20	40.46	7	2173	201.30	1409	1.54
PTC25	0.74	7.79	89.80	98.33	10	1060	187.00	1870	0.57
PTC26	0.05	28.32	312.50	340.85	3	3768	382.33	1147	3.29
PTC27	0.03	217.56	194.08	411.64	13	3051	137.15	1783	1.71
PTC28	0.55	53.06	24.06	77.22	7	2802	120.00	840	3.34
PTC29	1.69	8.83	128.98	138.88	10	680	99.10	991	0.69
PTC30	0.23	5.07	14.94	20.29	5	23231	200.80	1004	23.14
Average	2.86	23.56	119.45	145.82	5	3196	239.66	948	3.56

rank sum test [25] and the Vargha-Delaney's $\hat{A}_{12}$ effect size [120]. Both tests are non-parametric; therefore, they do not make any assumption on the data distributions. We used the Wilcoxon test to assess whether the difference in running time between the original test cases and the corresponding slices are statistically significant. For the sake of the analysis, we considered the cumulative execution time for all slices obtained for each individual system test case. We considered a level of significance $\alpha = 0.05$.

According to the Wilcoxon tests, the slices generated by DS3 have a statistically significant lower execution time than the corresponding non-sliced test cases (p-value=0.01). The Vargha-Delaney's statistic reports a medium effect size $\hat{A}_{12} = 0.69$.

DS3 Overhead As shown in the left side of table 5.2, DS3 takes, on average, 145.82 s to slice a complex system test case. The most time-consuming step is the refinement step, which recursively derives the hidden dependencies and performs backward slicing to guarantee that all related statements are included. This step takes, on average, 82% of the overall DS3 execution time. The second most expensive step is the generation of the initial set of slices using the static slicer; this step takes 16% of the overall DS3 execution time, on average. The remaining steps take, on average, only 2% of the total DS3 execution time.

It is worth noting that DS3 will be used just once to obtain the test case slices, so its overhead will be limited in any case. Further, using DS3 is particularly advantageous for those test cases that are executed many times a day, a common situation in continuous integration and deployment (CI/CD) environments, for example [104].

5.4.4 RQ3: Coverage and Fault Detection Capability

5.4.4.1 Methodology

To answer RQ3, we first compared the cumulative coverage of the original test cases and the test slices obtained through DS3. To measure code coverage, we used `Bullseye Coverage` [114], an advanced C++ code coverage tool used to improve software quality in critical system domains such as industrial control, medical, automotive, communications, aerospace, and defense. Next, we used mutation testing to assess the difference in fault detection capabilities between the original test cases and slices. Mutation testing is widely used in the literature to systematically assess the fault detection effectiveness of test cases [19, 45, 59], especially when not enough real-faults have been recorded, which is our case. Mutation testing introduces syntactic changes (mutations) into the production code using well-established mutation rules (mutation operators). The variants of the program produced by the mutation operators are often referred to as *mutants*. Effective test suites should pass on the original program but fail when executed against the *mutants*. In this scenario, the mutant is said to be *killed*; otherwise, the mutant is said to be *alive*.

For mutation analysis, we used `Mutate++` [73], an open-source mutation testing tool. `Mutate++` provides 15 mutations operators, including *arithmetic*, *conditional*, *Boolean*, *numeric*, and *line-deletion* operators. For our analysis, we selected six mutation operators based on the following observations. First, for some mutation operators, the majority of the generated mutants were killed at the build stage due to compilation errors; an example of such a mutation operator is the *line-deletion* operator, which removes a source code statement. Second, Lin et al. [71] reported that some mutation operators are sufficient in *selective mutation*. Selective mutation aims to reduce the number of mutants to consider without compromising the measurement of test effectiveness [53]. Based on the above observations, in our evaluation we considered the following mutation operators: (1) Logical Operator, (2) Conditional Operator, (3) Increment Decimal Operator, (4) Arithmetic Operator, (5) Boolean Literal Operator, (6) Decimal Number Operator.

To assess whether the slicing process of DS3 did not impact the fault detection capability of the test slices (group 1) compared to the original test cases (group 2), we compared the number of mutants killed by each group. We use K_O to denote the set of mutants killed by the original test cases and K_S to denote the set of mutants killed by the test slices generated by DS3.

We performed mutation testing and coverage analysis only on *JSBSim*. We could not consider *Prop* for the following reasons: (1) the proprietary system includes a large number of software components, written with different programming languages. Analyzing such a heterogeneous codebase would require a powerful and sophisticated mutation testing tool. (2) We did not have access to the system source code. (3) Running mutation testing on *Prop* would have required to run the test cases hundreds of times, once for each generated mutant. One execution for all *Prop* test cases requires ≈ 26 hours; hence, mutation testing for *Prop* would have been prohibitively expensive.

5.4.4.2 Results

Table 5.3 reports code coverage for the original test cases and the generated test slices. Columns “FunctionCov” and “BranchCov” indicate, respectively, function coverage and branch coverage scores; sub-columns “Total” indicate

Table 5.3: Comparison between original and sliced test cases in terms of code coverage

Subject	FunctionCov			BranchCov		
	Total	Original	Slices	Total	Original	Slices
JSBSim	2980	1831	1831	13541	5736	5698

Table 5.4: Comparison of the mutation testing results (K_O denotes the set of mutants killed by the original test cases; K_S denotes the set of mutants killed by the test slices produced by DS3)

Subject	ALL Mutants	K_O	K_S	$K_O \cap K_S$	$K_O \setminus K_S$
JSBSim	2678	289	288	288	1

the total number of functions/branches in the system source code; sub-columns "Original" and "Slices" indicate, respectively, the number of functions/branches covered by the original test cases and by the slices.

In terms of function coverage, both the original test cases and the generated slices achieved 61% (1831/2980). We observe a very small difference in branch coverage: the original test cases cover 42.36% (5736/13541) of the code branches whereas the generated slices covered 42.08% (5698/13541) of them.

We manually analyzed the test slices to understand the root cause of such (minor) differences in branch coverage. We observed that a few original test cases contain spurious function calls whose execution results are neither asserted nor used as input for other method calls. As such, these function calls do not contribute to the test scenario under test. Since DS3 uses an assertion-based slicing criterion, it can successfully identify these spurious statements as they are not included in any of the generated slices.

Table 5.4 shows the mutation testing results. For the original system test cases, 289 mutants were killed. The generated test slices were able to kill 288 mutants which, as expected, were all killed by the original test cases. To better understand why this single mutant was not killed by the generated slices, we manually analyzed the original (non-sliced) test case that killed that mutant as well as the slices generated by DS3. We found out that the mutant was injected within a function that was invoked by the original test case but removed in the generated slices. The function call was removed by DS3 because it does not contribute to the test cases' assertions; it is not used to create inputs for other method calls, and it does not access global resources (i.e., it does not have hidden dependencies). Though the original test case kills the mutant, it is not due to an assertion failure but rather a run-time error when invoking the function call after the mutant is injected. This negligible difference shows that *DS3 does not negatively impact the effectiveness in terms of code coverage and fault detection capability.*

5.4.5 Threats to Validity

Threats to *construct validity.* We evaluated DS3 using different metrics, namely (1) number of generated test slices with no compilation or run-time error, (2) running time, (3) code coverage, and (4) killed mutants. These metrics are widely used in testing [77,133]. To give a reasonable estimate of the test execution cost, we ran each test (both slices and original tests) 10 times and reported the average (arithmetic mean) results. To have a more reliable measure of the DS3 overhead, we also ran our approach 10 times.

Threats to *external validity.* We assessed DS3 in the context of a collaborative industrial research project with a large company in the aerospace domain. Hence, we reported the achieved results for one industrial, proprietary system. To improve the generalizability of our results and promote open science, we also included an open-source project, namely *JSBSim*, which implements a multi-platform, object-oriented Flight Dynamics Model written in C++.

5.5 Practical Implications

Test smells and maintainability We argue that DS3 contributes to addressing two test smells: *assertion roulette* and *eager tests. Assertion roulette* is a test with multiple assertion statements, which make root cause analysis more difficult in case of test failure [119]. *Eager tests* check multiple different functionalities at once, negatively affecting test code readability and understandability [119]. Executing DS3 on a given system test case will yield multiple slices, each with fewer assertion statements and one individual test scenario. Notice that the slice minimization stage further contributes to reduce the overall number of slices. Thanks to DS3, the assertions of the original test case will be distributed over the generated slices, thus reducing the number of assertions per slice (assertion roulette). Furthermore, splitting the test cases into independent slices helps address eager tests.

Regression testing Although DS3 has a non-negligible overhead (see Section 5.4.3), it is applied only once to generate the test slices. Therefore, its impact on the testing cost is limited. The advantage of using DS3 is that the generated slices are less expensive: this plays an important role in regression testing [133]. Reducing test execution time is one of the objectives of test case selection and prioritization. Using test slices, lightweight code analysis, and domain knowledge, test engineers can select and run the slices covering the changed or impacted portion of the production code, instead of the original test cases. In addition, test slices can be grouped by setup configuration, input values, or target functionalities. Slices can be further selected for each group to run only the most representative test slices and further reduce the overall test execution cost.

According to the results for RQ3, the test slices generated by DS3 have the same coverage and fault detection capability as the original test cases. Since the test slices are also statistically less expensive to run (see RQ2 results), which positively affects regression testing, there is a clear gain in using DS3.

Two common objectives used in test case selection and prioritization are coverage (to maximize) and test execution (to minimize) [133]. With DS3, developers can select test slices that reach the same coverage and mutation scores as the original test cases but with a significantly lower execution time.

5.6 Conclusion

In this chapter, we addressed the problem of dealing with complex system test cases containing multiple test scenarios, which negatively impact both regression testing and test evolution. We proposed DS3, a novel approach to decompose a complex system test case with multiple test scenarios into separate sliced test cases, each of them running one test scenario. DS3 leverages static slicing and the execution logs collected during past regression testing sessions. The main idea is to use logs containing run-time information about the SUT to identify dependencies between test statements due to the access and usage of global resources; these dependencies are used to refine sliced test cases generated by static slicing, which tend to miss such dependencies. The evaluation results, conducted on one proprietary system and one open-source system, show that log-based slice refinement is indeed effective at avoiding, in the generated sliced test cases, compilation or run-time errors due to missing dependencies. Furthermore, the generated test case slices are, on average, 3.56 times faster than the original system test case, with no significant loss in fault detection capability.

Chapter 6

Log-based Test Case Prioritization

Regression testing is performed during software evolution to have enough confidence that new software changes do not impact the behavior of the unchanged parts of the system. However, there could be insufficient resources to re-execute all existing test cases. Test case prioritization techniques improve the effectiveness of regression testing by finding an ordering of the test cases that maximizes a desirable property, such as the rate of fault detection.

Given that faults are unknown before executing the test suite, test case prioritization techniques use heuristics and software metrics that correlate with fault detection capabilities of test cases. Most of test case prioritization techniques use information like structural coverage, system models, requirements specifications, or mutation testing. Usually these information are extracted from the source code or the documentation of the target systems. However, in real-word systems, the visibility of the system behavior is limited to what is collected in its execution logs.

In this chapter we study the usage of test cases execution logs in the context of test case prioritization. Section 6.1 gives an overview of the challenges of test case prioritization; Section 6.2 illustrates our log-based test case prioritization tool; Section 6.3 presents the evaluation of our approach and Section 6.4 concludes this chapter.

6.1 Overview and Motivation

Regression testing is an important but expensive process. It is performed to have enough confidence that the introduced changes do not impact the behavior of the existing unchanged parts of the software [134]. Considering the limited testing resources and expensive test suites, test case prioritization techniques have been studied to allow for a cost-effective regression testing [18, 50]. Test case prioritization is challenging because it attempts to find an optimal ordering of test cases, before executing them, with the goal of maximizing the rate of fault detection. Specifically, given a test suite $TC = \{tc_1, tc_2, \ldots, tc_n\}$ composed of n test cases, a test case prioritization technique aims at finding an order (or permutations) of the test cases TC' that satisfy: 1) $f(TC') > f(TC)$, and 2) $f(TC') > f(TC'')$ for all other possible permutation $TC'' \neq TC'$, where f is a function used to measure the rate of fault detection of a sequence of test cases.

Given that faults are unknown before executing the test suite, heuristics and software metrics that correlate with fault detection capabilities of test cases have been proposed to tackle the challenges of test case prioritization. Most of the existing approaches rely on information collected from previous regression testing sessions like test case execution history [52,137] or structural coverage [31,74,133]. The historical information are used as prioritization criterion to estimate the fault detection capability of test cases. For example, the intuition behind using structural coverage is that early maximization of structural coverage will also increase the chance of early fault detection. However, such information is based on source code or documentation, which may not be available especially for third-party components.

Model-based test case prioritization techniques [50, 64, 107] were proposed to bypass the need for source code availability. Machine learning algorithms [20, 74] and mutation testing [54, 75] where mutants are used as surrogate for test case prioritization were also studied.

The existing techniques have certain benefits but also come with limitations. On one hand, one may think that accessing the source code may help achieve an effective ordering, however white-box techniques are considered expensive [82]. On the other hand, black-box techniques, although bypassing the source code availability requirement, have limited details about the target system, thus their effectiveness may be impaired.

Execution logs contain valuable information about the system states and behavior. A test case, when executed, reports a sequence of log messages that correspond to the log printing statements that were reached when executing the test case statements. Let SC be a system source code $SC = \langle s_1, \ldots, s_m \rangle$

with m lines of code or code statements. Some of these statements are log printing statements $PS \subset SC$, which means they will generate a defined log message (with or without variable values that change over executions) when they are reached (or executed). When executing a test case tc, it will produce a log file with a sequence of k log entries: $l = \langle e_1, \dots, e_k \rangle$, with $e_i = (t_i, mt_i, v_i)$, $t_i \in \mathbb{N}$ is the timestamp of the event occurrence, $mt_i \in MT$ is the message template that was produced by a specific log printing statement from PS, and $v_i \in V$ are the variable values that capture the system states, for $i = 1, \dots, k$ (for simplicity reasons, we will only consider the message templates in the rest of this chapter). In the same way a control flow graph shows the flows between code statements, the sequence of log entries capture the possible flows between log printing statements. These log printing statements reveal important information about the system state since, in practice, developers do not write log printing statements after each code statement, but they write them at important locations where they can get an idea about the changes of the system states. Since test cases logs reflect, at a certain level, the system status and they can be collected from previous executions, we find it interesting to study the usefulness of test cases logs in the context of test case prioritization.

In this chapter, we propose two different approaches, both relying on test cases logs collected from previous regression testing sessions. The first approach is a model-based approach where we build a system model [11, 13, 76] from the collected logs and then use it to compute certain model-based coverage [107]. Similar to structural coverage, test cases will be prioritized according to the number of nodes or edges that are covered by individual test cases. The difference is that the model we are using covers just the log printing statements and not the entire source code of the target system. The second approach is a machine learning approach where we build a regression model using different log-based features then we use this model to predict the number of mutants that will likely be killed by each test case. Test cases with the highest prediction values (the number of mutants to be killed) are considered to have high fault detection capability, thus will be executed first.

The intuition behind using execution logs in this context is that log printing statements are a subset of the system source code and they reflect a subset of the covered statements, therefore, we consider that early maximization of log printing statements coverage will increase the chance of early fault detection.

To summarize, the goal of this chapter is to investigate the usefulness of test cases logs in achieving an effective test case ordering. We propose a tool named LoTeCaP that utilizes test execution logs in two prioritization techniques: one based on the coverage of a log-based model, and the second uses a machine learning algorithm.

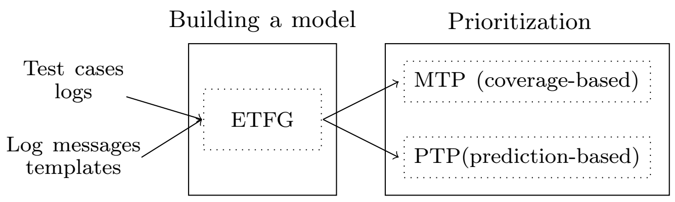

Figure 6.1: Overview of LoTeCaP

The main contributions of this chapter are:

1. a simplified log-based model that capture the system behavior from the executed log printing statements.
2. LoTeCaP, a tool for log-based test case prioritization that leverages the log-based model to measure model-based coverage and predict the mutants kills of each test case.
3. the evaluation of LoTeCaP in terms of fault detection rate.

6.2 LoTeCaP

In this section, we present our tool called *LoTeCaP* for Log-based Test Case Prioritization. We propose to build a log-based model from previously collected test case execution logs then leverage the model to prioritize test cases following two different techniques: the first technique is based on model coverage, and the second technique uses a machine learning algorithm to predict the mutant killing capability of test cases starting from log-based features to be extracted from the log-based model.

The rationale behind our approach is that execution logs contain valuable information of the system behavior. Additionally, logs describe a subset of the target system source code through the log printing statements which allows us to compute a portion of the code coverage. The challenges are: first how to model the system behavior using the execution logs in such a way that reflects the possible paths between the log printing statements and second how to use this model to prioritize test cases.

Figure 6.1 gives an abstract overview of the proposed approaches. The tool takes as input the test cases and their corresponding previously collected execution logs and the set of log message templates that are extracted from all log printing statements in the source code of the target system. The main step is to build a system model from the execution logs. The model we choose, named *Event Template Flow Graph* or *ETFG* for shirt, is like a control flow graph but only with a subset of the statements, which are the log printing statements. We propose two ways to use the log-based system model for test case prioritization: 1) compute model-based coverage (this approach is called Model-based Test case Prioritization, *MTP* for short), 2) a machine learning approach that leverages the log-based model to collect different features and use them to predict how many mutants a test case is likely to kill (this second approach is called Prediction-based Test case Prioritization, *PTP* for short).

In the following sub-sections we further describe the proposed approaches starting by the proposed log-based system model.

6.2.1 Log-based Model: ETFG

When a test case is executed, specific statements in the target program are being traversed. The log printing statements that were executed write messages to a log file. These log printing statements reveal important information about the system state since, in practice, developers do not write log printing statements after each code statement, but they write them at important locations where they can get an idea about the system state. Some log message templates can be repeated along the log file which means that the same log printing statement was traversed many times. Log messages can be seen as an excerpt of the target system source code: they show the possible sequences of log printing statements. To reconstruct the possible paths connecting log printing statements, we suggest to build a log-based model from test cases logs. Two important assumptions to consider: 1) each log printing statement prints a unique template. This is important when comparing the paths covered by two different test cases. 2) different executions of the same test case generate the same sequence of templates. Since log messages will be used to describe the test case, the sequence of log printing statements should be the same for each test case (no flaky tests).

Let $Logs = \{l_1, l_2, \ldots, l_n\}$ be the set of logs that were generated from the set of test cases $TC = \{tc_1, tc_2, \ldots, tc_n\}$), and each execution log is composed of a sequence of log entries $l_i = \langle (t_1, mt_1), \ldots, (t_k, mt_k) \rangle$ (as mentioned above, we ae not considering the variable parts of log messages). We present *ETFG*, the Event Template Flow Graph that summaries all possible paths between log printing statements as observed in all test cases logs. $ETFG = (N, E)$ is a directed graph where N is a set of nodes and E is a set of edges. N is defined by the set of templates occurring in all the test cases logs $Logs$ with

Algorithm 9 Build the Event Template Flow Graph ($ETFG$)

```
Input: Test cases executions logs $Logs = \{l_1, l_2 \ldots, l_n\}$
Output: Sets $Nodes$ and $Edges$ of the $ETFG$ model
 1: Set $Nodes \leftarrow \emptyset$
 2: Set $Edges \leftarrow \emptyset$
 3: for Log $l \in Logs$ do
 4:     for Message template $mt \in l$ do
 5:         $Nodes \leftarrow Nodes \cup mt$
 6:         $next \leftarrow GET-NEXT-MESSAGE-TEMPLATE(l, mt)$
 7:         $Edges \leftarrow Edges \cup (mt, next)$
 8:     end for
 9: end for
10: return Nodes , Edges
```

two special nodes ‘start‘ and ‘end‘. For all template mt_x occurring in ***Logs***, if mt_x is immediately followed by another template mt_y, then there is an edge $(x, y) \in E$ where x is the node of mt_x and y is the node of mt_y. Algorithm 9 presents the steps of building the *ETFG*. It uses the set of test cases logs and returns the *Nodes* and *Edges* of the *ETFG*. It starts with empty sets of *Nodes* and *Edges*, then it processes each log file of each test case to update the set of nodes with the newly observed message template *mt* (line 5). For each log file, each two consecutive message templates are considered an edge in the *ETFG* (lines 6-7).

This model is used to summarize and simplify the processing and manipulation of the test cases logs. The characteristics of this model are: it presents the paths, including conditional branches and loops, executed by the test cases as reported in the logs (the sequences of templates) and it is smaller in size than the control flow graph because it only captures the paths between log printing statements and not the entire source code statements.

The *ETFG* will be used in the model-based approach to compute model coverage, and will also be used in the machine learning approach to generate log-based features. The rationale behind these approaches is that, similar to structural coverage, log printing statements are a subset of the system source code so we can assess the effect of early fulfillment of log printing statements coverage on the rate of fault detection. Additionally, execution logs combine the static aspect of the source code and dynamic aspect of the system execution from which we may extract valuable data about test cases fault detection capability.

6.2.2 Model-based Test Case Prioritization

This first approach is a black-box approach where the only information we have about the test cases is the log printing statements that were traversed during their executions and reported in their log files. The *ETFG* of the test suite can be used as the target program model. We prioritize test cases according to certain model-based coverage criteria [107].

Given that the *ETFG* captures the flows between log printing statements, one can think of reusing and adjusting some of the well known model-based coverage criteria, such as additional node coverage and additional edge coverage. To measure the coverage of a test case tc_i on the *ETFG*, we traverse the *ETFG* using the log l_i produced by tc_i and keep track of the coverage properties. To prioritize the test cases, we iteratively select the test case tc_i that has the maximum additional coverage score compared to the coverage score achieved by the already prioritized test cases.

Some of the log messages have a severity level (e.g., WARN, ERROR). These severity levels are important in various activities like debugging because they indicate an issue or a possible issue within the system. We also leverage these details in the proposed log-based coverage criteria. We use the term high level nodes to refer to nodes of levels: *DEBUG, WARN, ERROR, FATAL.*

Model-based Coverage Criteria A test coverage criterion can be considered as a way of defining testing requirement to be fulfilled by test cases in order to maximize early fault detection during the testing activity. For a given test case tc_i and its generated log l_i, we define the following sets: $Nodes_i \subset N$ is the set of covered nodes by tc_i, and $Edges_i \subset E$ is the set of covered edges by tc_i

The basic model-based coverage criteria [107] can be applied on the *ETFG* graph: Additional Node coverage (*NodeCov*) and Additional Edge coverage (*EdgeCov*). During the prioritization process, we consider PTC as the sequence of ordered test cases. To count the number of additionally covered nodes (or edges) by a test case tc_i, we take into account the set SN (or SE) that contains the already covered/seen nodes (or edges) by test cases in PTC:

$NodeCov(tc_i) = |\{n \in Nodes_i \mid n \notin SN\}|$

$EdgeCov(tc_i) = |\{e \in Edges_i \mid e \notin SE\}|$

We suggest two variants of the edge coverage criterion that includes the logging level because we observed that high level nodes are usually placed in code blocks where there are complex computations and we considered that covering a high level node is likely to lead to a failure or a crash. We introduce the set of covered high level nodes for each test case $HighLevel_i \subset N$.

Additional Edge-Level coverage (*EdgeLevel)* prioritizes test cases that maximize the number of covered edges and maximize the number of covered high level nodes. The *EdgeLevel* criterion can be seen as a multi-objective function to maximize:
$max(\{e \in Edges_i \mid e \notin SE\})$ and $max(\{n \in HighLevel_i \mid n \notin SN\})$. During the prioritization process, at each iteration, the selected test case is the one that maximizes both conditions.

Additional Sum Edge-Level coverage (*SumEdgeLevel*) prioritizes test cases with the higher sum of covered edges and covered high level nodes.
$SumEdgeLevel(tc_i) = |\{e \in Edges_i \mid e \notin SE\}| + |\{n \in HighLevel_i \mid n \notin SN\}|$

We note that the *NodeCov* measures the covered log printing statements which is a subset of the statements coverage. The effectiveness of using this subset of statements for prioritization is discussed in the evaluation section.

6.2.3 Prediction-based Test Case Prioritization

Relying only on the number of covered nodes or edges in a system model is a basic way of characterizing test cases (we can say it is a static way for describing the execution paths of test cases). We believe that applying more sophisticated approaches on the log-based model to extract richer details about test cases can improve the effectiveness of log-based test case prioritization. Since our goal is to kill as many mutants as possible earlier during the regression testing activity, it is interesting to know if a test case will be able to kill mutants or not even before executing it. The log-based model *ETFG* presents the covered parts of the system source code in a concise way and facilitates the processing of the captured execution details (e.g., the covered nodes, the possible paths between nodes). We propose to apply a machine learning algorithm on test cases logs to predict the mutant killing capability of each test case. We consider two important assumptions: 1) historical data about the number of killed mutants per test case is provided and 2) the new faults/mutants introduced in the new version of the system should be similar to the mutants that were used to train the machine learning model. The key element of machine learning techniques is the features that are used to perform the prediction. We describe the extracted log-based features in the next paragraph.

Extracted Log-based Features We have identified different features from the execution logs that reflect how complex the execution path of a test case based on the traversed log printing statements.

1. BinaryCoveredNodes: we specify, for each test case, what nodes in the *ETFG* model were covered.

2. BinaryCoveredEdges: we specify, for each test case, what edges in the *ETFG* model were covered.

3. NumberCoveredLevels: we consider the different logging levels (e.g., DEBUG, INFO) that are observed in all test cases logs. For each test case, we count how many times a logging level was covered.

4. NodesVisits: we consider how many times a node was visited (different from the BinaryCoveredNodes where the values are binary for covered or not covered).

5. EdgesVisits: Similar to the NodesVisits, we consider the number of times each edge was visited by each test case.

6. NodesVisitsLevels: we combine the NodesVisits and the NumberCoveredLevels in this case.

7. EdgesVisitsLevels: we combine the EdgesVisits and the NumberCoveredLevels in this case.

To identify these features we used the *ETFG* model. As explained, the value to be predicted by our model (target) is the number of killed mutants by test case.

6.3 Evaluation

We implemented the *LoTeCaP* tool as a Python program. When using the program, one can specify what prioritization techniques to run depending on the available data and the time budget. Our experimental study is designed to answer five research questions.

RQ1: *How effective is MTP technique in early fault detection compared to the baselines?*

RQ2: *How effective are the different log-based coverage criteria in early fault detection?*

RQ3: *What is the impact of selecting hard-to-kill mutants on the effectiveness of the MTP approach in early fault detection?*

RQ4: *What regression model is best suitable for accurately predicting the number of killed mutants and with which features?*

RQ5: *How effective is PTP technique in early fault detection compared to the baselines and the MTP technique?*

The first three research questions assess the effectiveness of the model-based approach in test case prioritization. RQ1 compares the effectiveness of the MTP technique against the baselines (random, optimal and white-box prioritization) in terms of early fault detection. RQ2 assesses the impact of using different model-based coverage criteria in producing effective test cases orders. RQ3 studied the case when hard-to-kill mutants are introduced in the target program and assesses how effective the log-based model can be when used in prioritization. The fourth and fifth research questions evaluate the possibility of using the predicted number of killed mutants in test case prioritization. RQ4 investigates the possibility of applying machine learning on test cases logs to predict the number of mutants a test case is likely to kill. The last research question compares the effectiveness of MTP and PTP in early fault detection.

6.3.1 Benchmark

A candidate subject system should contain test cases that generate log files when executed. When a test case is executed, certain log printing statements in the SUT are traversed and generate log messages. Flaky test cases that generate different sequences of log messages over different executions are ignored because the behavior captured in the log will change and directly impact the prioritization outcome. Log messages templates (i.e., the strings contained in log printing statements) are needed to build the *ETFG*. It is important to make sure that each log printing statement contains a unique message. This is important to build an *ETFG* that correctly reflects paths between log printing statements as if the model was extracted from the CFG of the SUT source code. We used two open-source systems: Apache Kafka[1] a well known publish-subscribe-based messaging system, referred to as *Kafka*, having 7538 unit test cases and, *Evosuite*[2] having 1846 unit test cases. Both systems are written in Java and they use logging frameworks (Log4J, Logback). Such logging frameworks enrich log messages by adding the severity level.

To select the appropriate test cases for our evaluation, we executed all test cases multiple times: 1) to remove test cases that do not generate execution messages, 2) to identify flaky test cases (by comparing the resulted sequence of log messages templates between the different executions of the same test case). Log messages templates were collected from the source code by extracting all

[1]Apache Kafka: `https://kafka.apache.org`

[2]Evosuite: `https://www.evosuite.org`

logging statements (e.g., *logger.debug("executed function: foo()")*). We also checked the uniqueness of log printing statement messages. If a log message template is printed by more than one log printing statement, the template will be slightly modified (e.g., add a number or the name of the method to which the log printing statement belongs) in each of its corresponding log printing statements to account for the different positions of these log printing statements. We consider 894 test cases for *Kafka* and 248 test cases for *Evosuite*.

6.3.2 Mutants

We used the mutation testing tool PIT [24] to generate and execute mutants against the test suite under study. PIT provides a set of commonly used mutation operators [2,62] including Conditional Operator Replacement, Arithmetic Operator Replacement, and Relational Operator Replacement. We run PIT on the subject systems and collected the mutation testing reports that summarize the mutants killing status for each introduced mutant: for killed mutants, specify all test cases that killed each mutant. We use these details to assess the effectiveness of our proposed log-based test case prioritization approaches in early fault detection. We consider 996 mutants for *Kafka* and 701 mutants for *Evosuite*.

6.3.3 Evaluation Metrics

To assess the effectiveness in early fault detection of the proposed prioritization techniques, we used the well-known *APFD* score [36, 101]. It measures how fast a test suite can detect faults by calculating the weighted Average of the Percentage of Faults Detected over the life of a test suite. The *APFD* value ranges from 0 to 1. A higher score indicates that the first executed test cases are able to detect a high number of faults. Given a set of test cases $TC = \{tc_1, tc_2, \ldots, tc_n\}$ having n test cases and M different faults (or mutants) in the SUT, the *APFD* metric is calculated as follows:

$$APFD = 1 - \frac{P_1 + P_2 + \cdots + P_M}{n \times M} + \frac{1}{2 \times n}$$

where P_i is the position of the first test case that detected the fault i. This metric considers all faults as having the same severity.

To assess the accuracy of a regression model in predicting the number mutants to be killed by each test case, we use the Mean Absolute Error (*MAE*) score that represents the average value of error between the predicted values

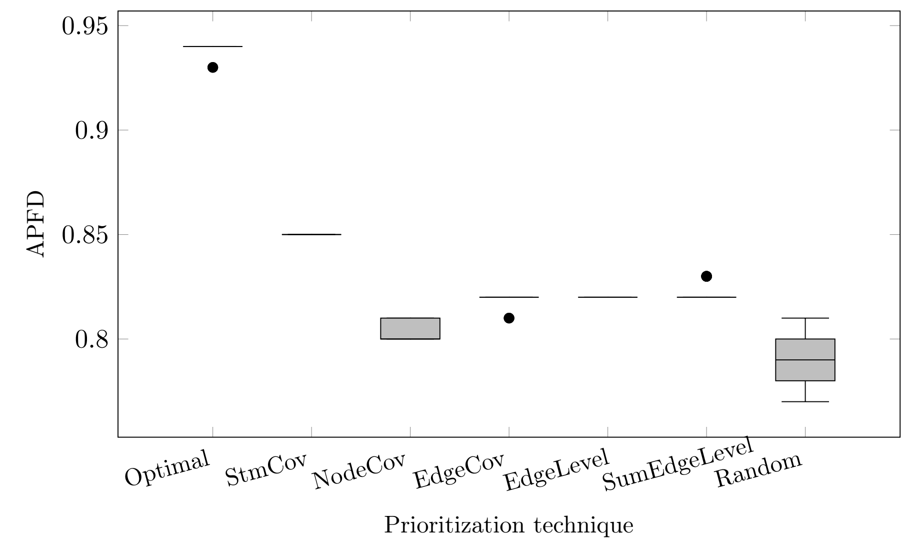

Figure 6.2: Comparison of the MTP approach and the baseline techniques applied on the Apache Kafka test cases.

and the oracle. *MAE* is computed as follows:

$$MAE = \frac{\sum_{i=1}^{S} |o_i - p_i|}{S}$$

where o_i is the actual output value, p_i is the predicted value, and S is the number of used data samples. Small values indicate that the predicted values are, on average, close to the expected (the real) values. In other terms, the regression model is able to predict accurate values.

6.3.4 RQ1: Effectiveness of MTP in Comparison with Baseline Techniques

In this first research question, our goal is to assess how effective is the MTP approach in early fault detection compared to baselines; white-box approach based on the statement coverage, random order and, optimal order (iteratively selects a test case that maximizes the number of additional killed mutants). We first used the test cases logs to build the *ETFG* model and then we used it to compute coverage: additional node coverage and additional edge coverage.

Figure 6.2 and in Figure 6.3 summarize the APFD scores of our MTP approach and the baseline prioritization techniques when applied on the *Apache*

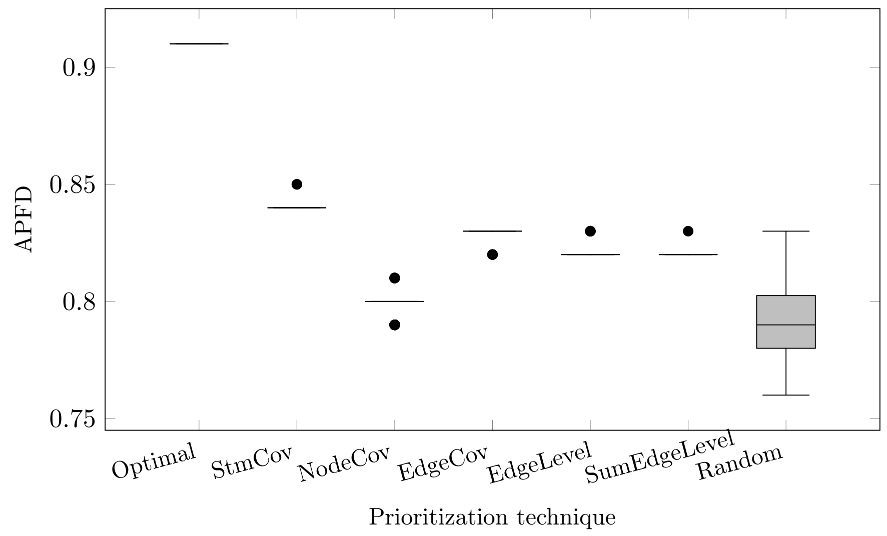

Figure 6.3: Comparison of the MTP approach and the baseline techniques applied on the Evosuite test cases.

Kafka and *Evosuite* test cases, respectively. We refer to "Optimal" for the optimal order, "StmCov" for additional statement coverage-based order, "NodeCov" for additional node coverage-based order, "EdgeCov" for additional edge coverage-based order, "EdgeLevel" for Additional edge-level coverage-based order, "SumEdgeLevel" for additional sum edge-level coverage-based order and, "Random" for random order.

We focus in this RQ1 on the first four box-plots and the random order box-plot. As shown in both figures, "NodeCov" and "EdgeCov" orders have, on one hand, a slightly higher APFD scores than the random order. This is because most of the introduced mutants were killed by many test cases. which means that when executing those test cases in almost any order, many mutants will be killed by the first test cases. On the other hand, "NodeCov" and "EdgeCov" have a lower but close APFD score compared to the "StmCov" order. This is explained by the fact that the *ETFG* model covers only log printing statements that constitute a small portion of all statements in the target system source code. The log-based model we built from *Kafka* test cases is composed of 389 nodes and 1427 edges, while the system has 87302 statements. For *Evosuite* system, the log-based model is composed of 364 nodes and 785 edges comparing to a total of 75931 statements in the system.

Despite the reduced number of covered statements by the *ETFG*, the ef-

Table 6.1: Comparison of prioritization effectiveness of the log-based coverage criteria. For every pair of criterion (C1, C2), we report the *p-value* based on the *Wilcoxon rank sum* test. The average *d* value is given to represent the *Cohen's d* effect size.

Pair		Kafka		Evosuite	
C1	C2	*p-value*	*d*	*p-value*	*d*
NodeCov	EdgeCov	< 2.2 e-16	−2.80	< 2.2 e-16	−8.08
NodeCov	EdgeLevel	< 2.2 e-16	−4.14	< 2.2 e-16	−5.80
NodeCov	SumEdgeLevel	< 2.2 e-16	−4.06	< 2.2 e-16	−6.13
EdgeCov	EdgeLevel	3.5 e-06	−0.70	< 2.2 e-16	3.80
EdgeCov	SumEdgeLevel	1.4 e-07	−0.80	< 2.2 e-16	5.08
EdgeLevel	SumEdgeLevel	0.03	−0.32	0.01	0.23

fectiveness of our approach is not badly affected compared to the white-box technique. This can be seen as a promising result for the usefulness of test cases logs in effective prioritization. Additional analysis of the distribution of log printing statements is required to investigate its impact on this testing activity.

6.3.5 RQ2: Comparing the Effectiveness of Model-based Coverage Criteria

In this second research question we compare the usefulness of model-based coverage criteria in obtaining effective test cases orders. We focus on *NodeCov*, *EdgeCov*, *EdgeLevel* and *SumEdgeLevel* orders.

From Figure 6.2 and in Figure 6.3, we notice that the APFD scores of the *NodeCov* criterion is lower than the other model-based criteria. Our explanation is that *EdgeCov* coverage subsumes *NodeCov* and it was proven to be more effective for test case prioritization. However the APFD scores from *EdgeLevel* criterion and *SumEdgeLevel* criterion are not that different from the *EdgeCov* criterion for both subject systems. Furthermore, we used the *Wilcoxon rank sum* to verify whether the APFD scores achieved by the model-based coverage criteria are significantly different, For this test, we consider a level of significance α=0.05. The Wilcoxon test is a non parametric test that is suitable for distributions with different variance. Additionally, we estimated

Table 6.2: Summary of the hard-to-kill mutants for each subject system.

Subject	All	Hard-to-kill		
		10%	1%	1
Kafka	996	972	644	184
Evosuite	701	621	204	113

the magnitude of the differences (effect size) using the *Cohen's d* statistic. The effect size is considered small when $d = 0.2$, and if $d = 0.8$ the effect is large.

Table 6.1 summarizes the statistical tests results. Column "C1" and "C2" indicate the pair of coverage criteria to compare, columns "*p-value*" and "*d*" present the *Wilcoxon* test results and the *Cohen's d* effect size, respectively. the *p-value* are smaller than the null hypothesis which implies that the effectiveness of the model-based coverage criteria are different. However, the effect size values are remarkably significant between the node coverage and the variants of edge coverage. To summarize, the edge coverage and its variants are significantly more effective in early fault detection than the node coverage criteria.

6.3.6 RQ3: Impact of Hard-to-Kill Mutants

In this research question, we challenge our MTP approach by using hard-to-kill mutants. We select three groups of mutants: 1) mutants killed by at most 10% of the test cases, 2) mutants killed by at most 1% of the test cases, and 3) mutants that are killed by just one test case. Table 6.2 summarizes the number of mutants in each group. Column "All" indicates the total number of mutants. Columns "10%", "1%" and, "1" indicate the number of mutants in each corresponding group.

Table 6.3 presents the effectiveness of the proposed techniques when applied on hard-to-kill mutants. Columns "10%", "1%" and, "1" report the average APFD scores of our log-based prioritization techniques when applied on; mutants killed by at most 10% of the test cases; mutants killed by at most 1% of the test cases; mutants that are killed by just one test case, respectively. As shown in Table 6.3, the effectiveness of the MTP approach decreases with hard-to-kill mutants. This can be explained by the fact that using the number of covered nodes or edges is a generic metric that misses fine-grained details about each test case execution. Relying only on the coverage of nodes or edges

Table 6.3: The effect of using hard-to-kill mutants on the effectiveness of the applied techniques in early fault detection. The average APFD scores are reported for each applied technique.

Technique	Kafka			Evosuite		
	10%	1%	1	10%	1%	1
Random	0.78	0.70	0.50	0.76	0.57	0.50
Optimal	0.93	0.91	0.90	0.89	0.81	0.77
StmCov	0.85	0.78	0.65	0.82	0.69	0.64
NodeCov	0.80	0.72	0.54	0.77	0.60	0.52
EdgeCov	0.81	0.73	0.56	0.80	0.62	0.55
EdgeLevel	0.82	0.74	0.56	0.80	0.62	0.54
SumEdgeLevel	0.82	0.74	0.56	0.80	0.64	0.59

is a basic way of characterizing test cases. Extracting additional details from execution logs could be beneficial for test case prioritization.

We investigate, in the next subsections, the usefulness of seven log-based features in predicting the mutant killing capability of test cases.

6.3.7 RQ4: Usefulness of Regression Models with Log-based Features

In this section, we study the possibility of predicting, for each test case, the number of killed mutants from the log-based model. The number of killed mutants will be used to correlate with the fault detection capability of test cases. To find the most adequate regression model in our context, we used two models: Linear Regression (*LR* for short) and Random Forest (*RF* for short). We used 10-fold cross validation technique to evaluate the accuracy of these models. as explained in Section 6.2.3, we used seven different log-based features. To evaluate the accuracy of the regression models, we used the MAE score that represents the average error between the predicted values and the true values.

Table 6.4 presents the accuracy of the used regression models in predicting the number of killed mutants per test case. Column "Features" indicate the applied log-based features. Column "*MAE-LR*" indicates the average error values of the linear regression model and, column "*MAE-RF*" indicates the

Table 6.4: Accuracy of the used regression models in predicting the number of killed mutants by each test case.

Features	*MAE-LR*		*MAE-RF*	
	Kafka	Evosuite	Kafka	Evosuite
BinaryCoveredNodes	7.00	29.05	4.53	18.79
BinaryCoveredEdges	19.24	49.97	4.44	19.27
NumberCoveredLevels	17.34	42.17	6.85	27.78
NodesVisits	8.21	39.29	4.37	18.92
EdgesVisits	19.13	48.86	4.34	19.11
NodesVisitsLevels	8.26	42.82	4.28	19.09
EdgesVisitsLevels	19.11	49.45	4.29	19.32

average error values of the random forest model. As shown in Table 6.4, the linear regression model has large MAE values compared to the random forest model for both subject systems. The largest error score for the linear regression model is when the model is trained with the *BinaryCoveredEdges* feature. For the random forest model, the largest error is when the model is trained with the *NumberCoveredLevels* feature. This means that these features are not well suitable to be associated with the mutant killing capability of test cases. The random forest model has the lowest mean error values: the *EdgesVisitsLevels* and the *NodesVisitsLevels* features produce accurate models with an average error 4.3 for Kafka data-set and 19.2 for Evosuite data-set. This means that the random forest model can predict the number of killed mutants with relatively small errors. We are interested in assessing how effective the test cases will be in early fault detection when prioritized using the predicted number of killed mutants.

6.3.8 RQ5: Comparing MTP and PTP

In this research question we aim at comparing the effectiveness of the ordered test cases based on the predicted number of killed mutants against the MTP technique and the baselines. To do so, we selected the Random Forest model and we used the *EdgeVisitsLevels* and the *NodesVisitsLevels* features to perform the prediction. To train the regression model, we used 70% of the data-set as training set and the other 30% for testing.

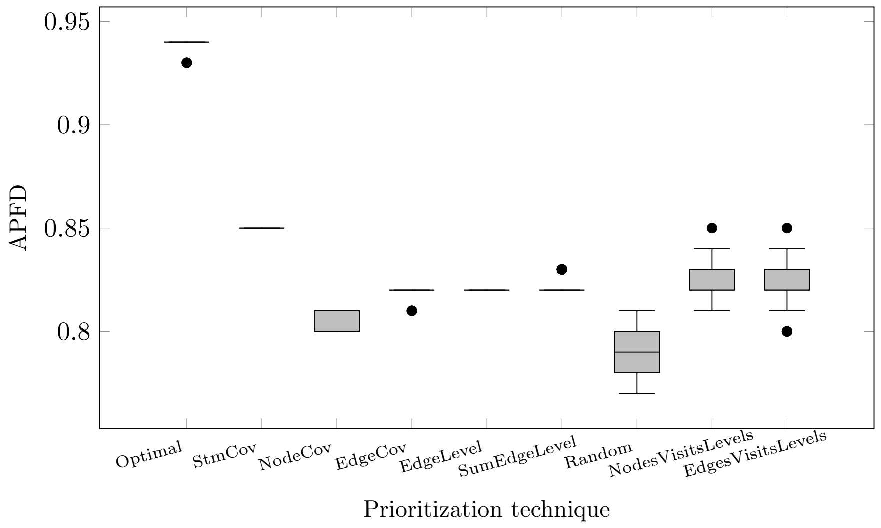

Figure 6.4: Comparison of the PTP and MTP approaches when applied on the Kafka test cases.

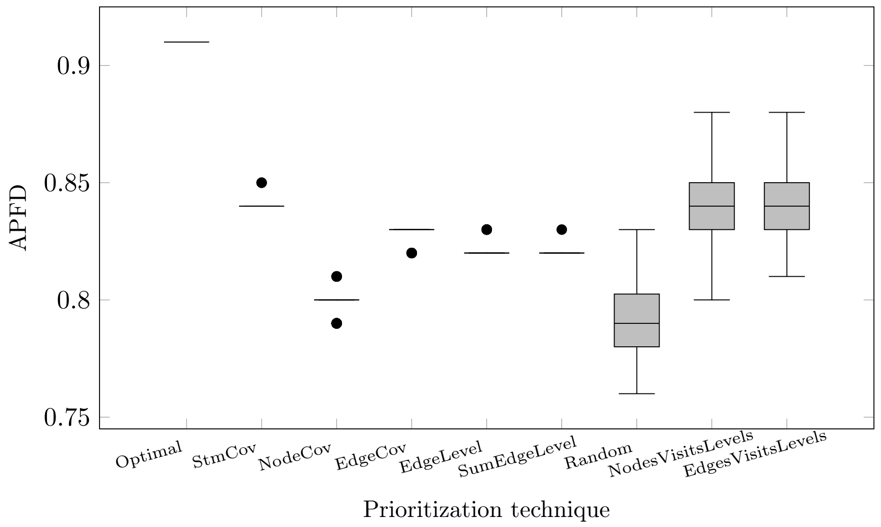

Figure 6.5: Comparison of the PTP and MTP approaches when applied on the Evosuite test cases.

Figure 6.4 and Figure 6.5 present the effectiveness of the obtained test cases orders that are generated by the both MTP and PTP approaches. With the Kafka test cases, the prediction-based prioritization resulted in test cases orders with similar effectiveness to the model-based technique. For Evosuite test cases, the APFD values of the prediction-based orders (using *EdgeVisitsLevels* and the *NodesVisitsLevels* features) are higher than the APFD scores of model-based orders. This is because the information that is used to correlate with the fault detection capability of test cases is more detailed and holds some dynamic characteristics of the dependencies between the nodes and edges of the log-based model.

Applying machine learning techniques with log-based features is a promising way to extract more informative and rich characteristics of test cases. Compared to code coverage, collecting execution logs is not an expensive nor difficult task. To summarize, execution logs are a rich and valuable source of information that can be applied to support software regression testing.

6.3.9 Threats to Validity

The logging granularity in terms of the locations of the log printing statements and the logging levels have a direct impact on the quality of the used features. If log printing statements are covered by all test cases, the feature will be too generic and will not catch the execution characteristics of each test case. Additionally, our prediction-based approach inherits the limitations of machine learning techniques. It the model is trained with poorly extracted features, the predicted results will definitely have a poor quality. In our case, the historical data related to the number of killed mutants from previous versions limits the prediction capability of the used regression model when applied on newer versions of the system with large differences in the code-base.

6.4 Conclusion

With the goal of having a cost-effective regression testing, we introduced a log-based tool, called LoTeCaP, to assess the usability of execution logs in the context of test case prioritization. We proposed two approaches, *MTP* and *PTP*, that leverage a log-based model to characterize the fault detection capability of test cases. *MTP* relies on model coverage and *PTP* predict the mutant killing capability of test cases using regression model that is trained from log-driven features. Despite the fact that the code coverage of log printing statements is small compared to statement coverage, our evaluation showed that log-based prioritization can result in effective test orders for early fault detection.

Chapter 7

Related Work

In this chapter, the related work are presented as follows: Section 7.1 highlights the related work to problem of log messages format identification (presented in Chapter 3); Section 7.2 illustrates the related work to model inference (presented in Chapter 4); and Section 7.3 Section 7.4 present the related work to regression testing cost reduction and test case prioritization (presented in Chapter 5 and Chapter 6, respectively).

7.1 Log Message Format Identification

Researchers have proposed various black-box techniques for the log message format identification problem. Most of these techniques rely on clustering [34, 48]. In [112] authors suggested *LogSig*, a cluster-based technique that requires a value *k* that determines the number of clusters to build. *LogSig* considers the case when variable values may have dynamic length. It starts by constructing a set of pairs of terms from each log message while it preserves the order between these terms in the original log messages. Next, it creates *k* random clusters of log messages and then starts moving log messages from one cluster to another in order to obtain better representative clusters. *LogSig* uses a local search strategy to measure, what they called, “purity” of term pairs. This measure is computed for each cluster: it is maximized when every pair of terms is contained in every log message. This measure reflects the number of common pairs of terms between log messages. Finally, *LogSig* builds the event types leveraging the most frequent common term pairs in the final clusters. This technique relies on the user-defined value *k* which directly affect the quality of the obtained message templates. Another clustering technique was presented in [41]: *Log Key Extraction (LKE)* . It is a mixture of hierarchical clustering and heuristic rules. The technique removes trivial dynamic fields using pre-defined rules to obtain raw log keys. The clustering step groups raw log keys using a weighted edit-distance metric. This technique proposes a group splitting algorithm that further splits the initial clusters to ovoid merging different event types. The last step extracts the common words from each final clusters and construct the event templates. *LKE* required an effort to fine tune its three parameters: the edit distance weight, a content threshold to determine if a word in the log message to be considered fixed or not and, a cluster threshold based on the k-means technique. The *SLCT* technique was presented in [116] as a three-step density-based approach. It makes a first pass over the log file to build a word vocabulary based on word frequency. In the seconds pass, *SLCT* creates candidate clusters using the data summary. The last step consists of selecting candidate clusters having more log messages than the user-defined threshold (frequent words are most likely to represent the fixed part of the log message template) and abstract them into event types. *SLCT* is prone to overfitting with small user-defined threshold values. *IPLoM* [78] is an iterative partitioning technique. The first partitioning is based on the log messages length (number of tokens). The second partitioning relies on positions of tokens with less unique values (tokens with less unique values are likely to be fixed tokens). The final partitioning uses a heuristic to find bijective relationships between tokens at the same position to further split the clusters. Once the hierarchical partitioning is done, *IPLoM* counts the number of unique values of each token in the obtained partitions. A token position with only one unique value is considered a fixed word otherwise it is a variable word. A limitation of this technique appears when considering a fixed word to only have one unique value. This may merge different event formats, hence resulting in inaccurate message templates. Additionally, it requires fine tuning of five parameters.

AEL [55] (Abstracting Execution Logs) is a heuristic-based approach. It follows a similarity detection heuristic to abstract log messages. This technique is composed of four main steps. First, an anonymization step is applied to identify dynamic fields (e.g., numbers, IP addresses). The anonymization uses domain-experts knowledge to specify the format of the dynamic fields. Next, the anonymized log messages are divided into clusters: log messages having the same number of words and the same number of estimated variable fields will be in the same cluster. Then, for each cluster, *AEL* compares each

message will all other messages in order to find the most suitable event abstraction. This step is called categorization. Finaly, *AEL* performs a reconciliation step to examine and fix the obtained events because the anonymization step may not correctly identify dynamic fields. Similar to our approach, *AEL* does not have parameters to be tuned but it requires domain experts to hard-code rules to identify dynamic fields in raw log messages.
A technique using NLP is presented in [63]. It applies a supervised natural language processing technique (Conditional Random Fields) to classify words as a fixed/variable parts. This technique uses regular expressions based on domain knowledge for data labeling. Another NLP-based technique is proposed in [115]. This technique uses a character-based neural network to classify fixed/variable parts of log message. Those supervised NLP-based techniques have some limitations: the need for domain knowledge to label data and they require large logs to train accurate models.
Other researchers were focusing on online log parsing [33,49,90]. Online techniques are useful when a large volume of logs must be processed in a streaming manner for monitoring purpose for example. *DRAIN* [49] leverages a fixed depth parse-tree to guide the log format extraction. *DRAIN* uses a pre-processing step to identify trivial variable fields using regular expressions. *DRAIN* build a parse tree with first layer consisting of groups of log messages, each group represent a different length (number of words). It considers words at the beginning of the log messages are likely to be fixed and It measures a similarity score between the log message in each group to identify the best suitable event formats. The efficiency of online log parsing techniques depends on the characteristics of the processed data (e.g., number of log messages, length of log messages, ...) and they have high memory consumption.
An empirical study carried in [48] comparing four techniques for the log message format identification problem: *SLCT* [117], *LKE* [41], *LogSig* [113], and *IPLoM* [78]. The results of this study revealed that (i) IPLoM produces the most accurate templates, and (ii) log pre-processing (like the one applied in MoLFI) is very critical to achieving good clustering performance. Compared to *DRAIN*, *IPLoM* generates incorrect templates. One limitation of the two best techniques (*DRAIN* and *IPLoM*) is that they require their parameters to be tuned for each log file. Such parameters, if not chosen carefully, will significantly affect the performance of the tool. Different from state-of-the-art techniques, our approach MoLFI uses an automated heuristic to automatically choose the best compromise between the two objectives of the log message format identification problem (frequency and specificity) without using a priori, user-defined thresholds.

7.2 Log-based Model Inference

Starting from the seminal work of Biermann and Feldman [13] on the *k-Tail* algorithm, which is based on the concept of state merging, several approaches have been proposed to infer a Finite State Machine (FSM) from execution traces or logs. *Synoptic* [11] uses temporal invariants, mined from execution traces, to steer the FSM inference process to find models that satisfy such invariants; the space of the possible models is then explored using a combination of model refinement and coarsening. *InvariMINT* [9] is an approach enabling the declarative specification of model inference algorithms in terms of the types of properties that will be enforced in the inferred model; the empirical results show that the declarative approach outperforms procedural implementations of *k-Tail* and *Synoptic*. Nevertheless, this approach requires prior knowledge of the properties that should hold on the inferred model; such a pre-condition cannot be satisfied in contexts (like the one in which this work is set) where system components are black-boxes and the knowledge about the system is limited. Other approaches infer other types of behavioral models that are richer than an FSM. *GK-tail+* [80] infers guarded FSM (gFSM) by extending the *k-Tail* algorithm and combining it with Daikon [38] to synthesize constraints on parameter values; such constraints are represented as guards of the transitions of the inferred model. *MINT* [123] also infers a gFSM by combining EDSM (Evidence-Driven State Merging) [21] and data classifier inference [89]. EDSM, based on the Blue-Fringe algorithm [70], is a popular and accurate model inference technique, which won the Abbadingo [70] and the StaMinA competition [122]. Data-classifier inference identifies patterns or rules between data values of an event and its subsequent events. Using data classifiers, the data rules and their subsequent events are explicitly tied together. *ReHMM* (Reinforcement learning-based Hidden Markov Modeling) [37] infers a gFSM extended with transition probabilities, by using a hybrid technique that combines stochastic modeling and reinforcement learning. ReHMM is built on top of MINT; differently from the latter, it uses a specific data classifier (Hidden Markov model) to deal with transition probabilities. All the aforementioned approaches cannot avoid scalability issues due to the intrinsic computational complexity of inferring FSM-like models; the minimal consistent FSM inference is NP complete [43] and all of the practical approaches are approximation algorithm with polynomial complexity.

Model inference has also been proposed in the context of distributed and concurrent systems. *CSight* [10] infers a communicating FSM from logs of vector-timestamped concurrent executions, by mining temporal properties and refining the inferred model in a way similar to *Synoptic*. *MSGMiner* [68] is a framework for mining graph-based models (called Message Sequence Graphs) of distributed systems; the nodes of this graph correspond to Message Sequence Chart, whereas the edges are determined using automata learning techniques. This work has been further extended [69] to infer (symbolic) class level specifications. However, these approaches require the availability of channel definitions, i.e., which events are used to send and receive messages among components.

Liu and Dongen [72] uses a *divide and conquer* strategy, similar to the one in our SCALER approach, to infer a system-level, hierarchical process model (in the form of a Petri net with nested transitions) from the logs of interleaved components, by leveraging the calling relation between the methods of different components. This approach assumes the knowledge of the caller and callee of each component methods; in our case, we do not have this information and rely on the *leads-to* relation among log entries, computed from high-level architectural descriptions and information about the communication events.

One way to tackle the intrinsic scalability issue of (automata-based) model inference is to rely on distributed computing models, such as MapReduce [28], by transforming the sequential model inference algorithms into their corresponding distributed version. In the case of the *k-Tail* algorithm, the main idea [125] is to parallelize the algorithm by dividing the traces into several groups, and then run an instance of the sequential algorithm on each of them. A more fine-grained version [76] parallelizes both the trace slicing and the model synthesis steps. Being based on MapReduce, both approaches require to encode the data to be exchanged between mappers and reducers in the form of key-value pairs. This encoding, especially in the trace slicing step, is application-specific; hence, it cannot be used in contexts in which the system is treated as a black-box, with limited information about the data recorded in the log entries. Furthermore, though the approach can infer a FSM from large logs of over 100 million events, the distributed model synthesis can be significantly slower for $k \geq 2$, since the underlying algorithm is exponential in k.

7.3 Test Case Slicing

The closest approaches to DS3 are those based on *test carving* techniques [35, 60], which automatically carve unit test cases from system test cases. Such techniques consist of capturing, for a specific target unit method, the system states before (pre-state) and after (post-state) the invocation of the unit method during the execution of the system test case. From the pre-state, the unit method is replayed and the resulting state is queried to determine if there are differences with the recorded post-state. One of the main differences between these techniques and ours is that DS3 preserves the system level characteristics of the obtained (sliced) test cases (which can potentially entail a very complex usage of the various units), whereas carved tests target unit methods. Another difference is that carving techniques require to instrument the program code to capture pre- and post-states and to replay the unit method, whereas DS3 relies on program slicing and on information, regarding global resource usage, recorded in execution logs.

[131] propose *test case purification* for improving fault localization, by separating existing test cases into small fractions (called purified test cases). Similar to DS3, they use program slicing on assertions in an original test case to generate single-assertion test cases. Since they mostly target inexpensive, unit test cases (rather than expensive system test cases), dynamic slicing represents a viable solution for their approach. In contrast, DS3 deals with system test cases, relies on static slicing extended with log-based refinement to capture all dependencies in test case statements, and does not require the execution of test cases. [4] propose to extract unit test cases from system test cases through the reverse execution (called time travel debugging) of a program flow for reconstructing object creation and modification statements from the source code. The approach also uses differential analysis to identify the test statements from which the unit test case will be extracted. Different from DS3, which requires only access to the test cases and to the execution logs, this approach requires access to the source code of the SUT. Finally, [57] investigated whether coarser granularity tests could be automatically generated by aggregating unit tests using Differential Unit Tests (DUT), initially developed for test carving by [35]. Such a strategy is, conceptually speaking, dual to test case decomposition.

DS3 represents an enhancement of vanilla *static* slicing [126]; as discussed in section 5.2, the latter is most likely to miss hidden dependencies among statements, originated from the usage of global resources (e.g., external files) within the test case program. Compared to vanilla *static* slicing, DS3 requires one single execution of the test cases (from previous regression testing activities) to collect and parse the execution log files.

Dynamic slicing [65] is another form of enhancement of static slicing, which considers only specific executions of the program for a given objective (e.g., to perform debugging and root cause analysis). Dynamic slicing requires running test cases and accessing the source code for instrumentation and coverage analysis. Slices can be generated by analyzing the execution paths and the dependencies across the executed statements. However, this alternative is not feasible for a system composed of third-party components, and it does not handle "hidden" dependencies. Indeed, code coverage does not include information about which external resource has been accessed during the test execution. [14] presents a particular type of dynamic slicing technique, called *observation-based* slicing, which aims to slice programs independently from the programming language used. Although this technique can be used for multi-language systems that include (3rd-party) binary components, it requires a large number of executions of the test case under analysis to iteratively slice candidates and check their validity (i.e., to execute the obtained slice to make sure it runs without compilation or run-time errors). This requirement makes observation-based slicing impractical for complex system test cases (especially those in the cyber-physical system domain [1, 47]), whose execution is time-consuming. For this type of systems, DS3 is preferable as it does not require additional test case executions for assessing the validity of the generated slices.

A side benefit of the application of DS3 is the removal of some test code smells [83]. In this sense, DS3 is related to approaches for (test) code smells refactoring [81, 119], which introduced catalogues of test smells together with (manual) refactoring operations to address them. Based on existing catalogues, there have been proposals [93,97] to automatically detect (rather than refactor) test smells. These approaches rely on detection rules that raise warnings when some metrics (e.g., size, number of method calls, number of assertions) in the test code exceed given thresholds. However, a recent study [96] showed that detection rules are far less accurate than previously reported, especially when tests use external resources.

Although the primary goal of DS3 is neither to detect nor to refactor test smells, by slicing test cases into separate sliced test cases with fewer assertions, DS3 addresses the *eager tests* and *assertion roulette* test smells. Furthermore, eager tests are identified through static slicing and log analysis, instead of using detection rules, as proposed in the literature.

7.4 Test Case Prioritization

The test case prioritization approaches proposed in our tool LoTeCaP belong to the group of black-box model-based test case prioritization techniques [66]. An EFSM model build from the system specifications was used in [67]. To perform test prioritization, the original system model and the modified one are used to identify a set of elementary model modifications (such as a transition addition or a transition deletion). They present two model-based test prioritization techniques: 1) selective test prioritization where a high priority is assigned to tests that execute one of the modified transitions in the modified system model while the other tests (that do not cover any modified transition) will have a low priority, 2) a model dependence-based test prioritization where they apply model dependence analysis to identify the possible interactions between the modified transitions and the other parts of the system model. The main difference with our model-based technique is that they use the system specifications to build the system model while we simply use test cases logs from previous executions. In the context of Combinatorial Interaction Testing, other researches use a model of the system inputs to prioritize test cases [50, 98] by maximizing the interactions between model inputs. In the context of software product line testing, authors in [51] mutate the constraints of the input model then prioritize test cases based on the number of model mutants they kill.

The idea of using execution logs for test case prioritization is studied, to some extent, in the context of behavior regression testing [39, 88, 129]. The work presented in [79] uses execution traces of a component-based system to generate behavioral models in the form of finite state machines labeled with method invocations. They use the kBehavior engine for model inference. The derived models represent aspects of component integration like exchanged data and interfaces invocations. Test cases are grouped based on the interactions of the system with the tested component, then they are ordered according to the complexity of the triggered interactions. The main differences between this technique and ours are the type of the inferred model and the usage of the model in the prioritization process. Based on our experience, the kBehavior engine has scalability issues and a low accuracy score in terms of trace acceptance. However our *ETFG* is a subset of the control flow graph which preserves the static aspect of the source code to make the graph simpler and easier to produce. *ETFG* takes advantage of its similarity with the control flow graph and reuses some of the well studied model-based coverage criteria, which have been proven effective in many studies. Another tool for automated behavioral regression testing, called BERT, was presented in [56]. First the tool executes the same test suite on two versions of the code using test inputs that focus on the changed parts of the code. Next, it identifies behavioral differences between the two versions using dynamic analysis. The approach is related to our work in the way the tool analyzes and orders the behavior differences. The analysis will help the developers assess which modification may cause a regression fault. However, this tool has a number of limitations: first, it mainly relies on the source code which is not always available, second, the tool is designed to work on localized changes involving few classes and may not be effective with changes that add new functionalities to a system.

LoTeCaP is also related to mutation-based techniques where instead of using model coverage of test cases, mutants are used as surrogate for test case prioritization [32, 75]. The work presented in [136] is considered the first mutation testing approach to predict the result of mutation testing (mutant is killed or survived) without executing the mutants against the test suite. Their approach is based on a classification model using different features from previously executed mutants of earlier versions of the project. Differently from their approach, we use a regression model to predict the number of possibly killed mutants for each test case and not the status of each individual mutant. Additionally, the features we used are solely based on previous execution logs of the test cases. In [106], the authors investigate a new mutation-based test case prioritization approach that relies on distinguishing the behavior of individual mutants from each other and that of the original program. The proposed mutation-based prioritization technique in [75] relies on the difference between the early version and the latter version of the target system. It takes as input the source code of two versions of the system, then, as a first step, it identifies the differences in terms of statements between the two versions of the source code. The second step is to generate a set of mutants whose mutation operators occur in the source code difference. The third step is to calculate the fault detection capability of each test case using two different methods: a statistics-based model that relies on the number of killed mutants per test case because the authors consider that the more mutants a test case kills, the higher the probability of fault detection would be, and a probability-based model that calculates the fault detection capability based on the distribution of mutants over the statements difference identified in the first step. Differently from their approach, LoTeCaP does not require the source code and the only artifacts we need are the test cases logs and historical information about the number of killed mutants by each test case.

Machine learning algorithms were also applied to achieve cost-effective regression testing [74]. The technique proposed in [20] aims at accelerating compiler testing by leverages the characteristics of test cases that trigger bugs to learn two models: a capability model to predict the likelihood of a new test case for triggering bugs and, a time model to predict the execution time of each test case. This technique extracts features from the source code of test cases (e.g., pointer comparison features, address features) which may be expensive for the case of cyber-physical systems. The work in [110] presented a lightweight method for test case prioritization and selection in Continuous Integration that combines reinforcement learning with historical test information like indicators of failing test cases. Existing machine-learning-based techniques have not taken advantage of potentially available and useful data.

The features used by most papers are limited to readily available data, such as code complexity feature or code coverage features. There is lack of tools that include logs in the prioritization decision [94], as a result, potentially relevant features remain unused. Our tool LoTeCaP includes another source of data into the test case prioritization process by leveraging previously recorded execution logs to extract new features.

Chapter 8

Conclusions & Future Work

8.1 Summary

The goal of this thesis is to investigate the usefulness of system execution logs in supporting different software engineering tasks. We have addressed several challenges when processing and analyzing execution logs of cyber-physical systems.

In Chapter 3 we formulated the *log message format identification problem* problem as a multi-objective optimization one, where the goal is to generate log message templates with high frequency (i.e., they match as many log entries as possible) and high specificity (i.e., specific for each log event). To tackle the problem, we introduced MoLFI, a tool implementing a search-based approach based on a multi-objective genetic algorithm and trade-off analysis. An empirical study involving six real-world datasets (five publicly-available and one proprietary) showed that MoLFI (i) achieved significantly higher accuracy than DRAIN and IPLoM, two state-of-the-art tools; (ii) is highly scalable to large logs since it requires slightly above two minutes to analyze hundreds of thousands of messages.

In Chapter 4 we addressed the scalability problem of inferring the model of a component-based system from the individual component-level logs, assuming only limited (and possibly incomplete) knowledge about the system. Our approach, called SCALER, first infers a model of each system component from the corresponding logs; then, it merges the individual component models together taking into account the dependencies among components, as reflected in the logs. Our evaluation, performed on logs from an industrial system, has shown that SCALER can process larger logs, is faster, and yields more accurate models than a state-of-the-art technique.

In Chapter 5 we addressed the problem of dealing with complex system test cases containing multiple test scenarios, which negatively impact both regression testing and test evolution. We proposed DS3, a novel approach to decompose a complex system test case with multiple test scenarios into separate sliced test cases, each of them running one test scenario. DS3 leverages static slicing and the execution logs collected during past regression testing sessions. The main idea is to use logs containing run-time information to identify dependencies between test statements due to the access and usage of global resources; these dependencies are used to refine sliced test cases generated by static slicing, which tend to miss such dependencies.

In Chapter 6 we investigated the possibility of using test cases logs to perform test case prioritization. We presented a tool, called *LoTeCaP*, that proposes two log-based approaches for test case prioritization. A model-based technique, called *MTP*, where we build a model using test cases logs and then used it to compute different model-based coverage criteria: we propose two new criteria combining the logging level with edge coverage. And a second approach, called *PTP*, based on a machine learning technique. We used log-based features to train a regression model and then predict the number of mutants that each test case is likely to kill and then we used the predicted values to order the test cases. Logs can indeed be used to extract useful information to support regression testing.

8.2 Future Work

As part of future work, we plan to improve the proposed log-based techniques and assess their usefulness in other software engineering tasks. For MoLFI, we plan to investigate other encoding schemas, experimenting with other formulations of the problem (e.g., by introducing the coverage of log messages as another optimization objective), and by handling semantically equivalent templates. We also plan to assess the use of MoLFI for supporting various software maintenance and testing activities, such as boosting test case generation techniques through the definition of new seeding strategies [100] based on input and output values (variable parts) observed in the logs. For SCALER, we are working on refining the heuristics used for identifying the dependencies of the log entries between multiple components, to take into account logs with imprecise timestamps and out-of-order messages. We also plan to evaluate SCALER on different data-sets and to integrate it with other model inference techniques and assess the effectiveness of the inferred models in software engineering activities, such as test case generation. Related to DS3, we plan to assess the impact of the sliced test cases on the cost-effectiveness of regression testing activities, such as test case prioritization. And finally for LoTeCaP, we plan to study other machine learning algorithm on test cases logs to support regression testing. We plan to investigate other log-based features. Furthermore, we will apply our tool for log-based test case prioritization on industrial case studies with additional execution constraints.

Bibliography

[1] Abdessalem, R. B., Panichella, A., Nejati, S., Briand, L. C., and Stifter, T. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)* (New York, NY, USA, 2018), IEEE, pp. 143–154.

[2] Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A. S. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering 32*, 8 (2006), 608–624.

[3] Arcuri, A., and Fraser, G. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering 18*, 3 (2013), 594–623.

[4] Bach, T., Pannemans, R., Haeussler, J., and Andrzejak, A. Dynamic unit test extraction via time travel debugging for test cost reduction. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (New York, NY, USA, 2019), ACM, pp. 238–239.

[5] Baker, R. D. Modern permutation test software. In *Randomization Tests, Third Edition*. Marcel Dekker, New York, NY, USA, 1995, pp. 391–401.

[6] Basin, D., Caronni, G., Ereth, S., Harvan, M., Klaedtke, F., and Mantel, H. Scalable offline monitoring. In *Proceedings of the 5th*

International Conference on Runtime Verification (RV 2014) (Cham, Switzerland, 2014), vol. 8734 of *LNCS*, Springer, pp. 31–47.

[7] Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., and Binkley, D. Are test smells really harmful? an empirical study. *Empirical Software Engineering 20*, 4 (2015), 1052–1094.

[8] Bertero, C., Roy, M., Sauvanaud, C., and Trédan, G. Experience report: Log mining using natural language processing and application to anomaly detection. In *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE 2017)* (Piscataway, NJ, USA, 2017), IEEE, pp. 351–360.

[9] Beschastnikh, I., Brun, Y., Abrahamson, J., Ernst, M. D., and Krishnamurthy, A. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. Softw. Eng. 41*, 4 (2015), 408–428.

[10] Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)* (New York, NY, USA, 2014), ACM, pp. 468–479.

[11] Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., and Ernst, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)* (New York, NY, USA, 2011), ACM, pp. 267–277.

[12] Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., and Ernst, M. D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011)* (New York, NY, USA, 2011), ACM, pp. 267–277.

[13] Biermann, A. W., and Feldman, J. A. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput. 100*, 6 (1972), 592–597.

[14] Binkley, D., Gold, N., Harman, M., Islam, S., Krinke, J., and Yoo, S. Orbs: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), Association for Computing Machinery, pp. 109–120.

[15] Branke, J., Deb, K., Dierolf, H., and Osswald, M. Finding knees in multi-objective optimization. In *Proceedings of the 8th International Parallel Problem Solving from Nature (PPSN 2004)* (Berlin, Heidelberg, 2004), vol. 3242 of *LNCS*, Springer, pp. 722–731.

[16] Briand, L. C., Labiche, Y., and Leduc, J. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Trans. Softw. Eng. 32*, 9 (2006), 642–663.

[17] Briand, L. C., Labiche, Y., and Shousha, M. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. *Genetic Programming and Evolvable Machines 7*, 2 (2006), 145–170.

[18] Catal, C., and Mishra, D. Test case prioritization: a systematic mapping study. *Software Quality Journal 21*, 3 (2013), 445–478.

[19] Chekam, T. T., Papadakis, M., Traon, Y. L., and Harman, M. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)* (New York, NY, USA, 2017), IEEE, pp. 597–608.

[20] Chen, J., Bai, Y., Hao, D., Xiong, Y., Zhang, H., and Xie, B. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)* (2017), IEEE, pp. 700–711.

[21] Cheng, K., and Krishnakumar, A. S. Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th Design Automation Conference (DAC 1993)* (New York, NY, USA, 1993), ACM, pp. 86–91.

[22] Clarke Jr, E. M., Grumberg, O., Kroening, D., Peled, D., and Veith, H. *Model checking.* MIT press, 2018.

[23] Cobb, H. G., and Grefenstette, J. J. Genetic algorithms for tracking changing environments. In *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA 1993)* (San Francisco, CA, USA, 1993), Morgan Kaufmann Publishers, pp. 523–530.

[24] Coles, H., Laurent, T., Henard, C., Papadakis, M., and Ventresque, A. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis* (2016), pp. 449–452.

[25] Conover, W. J. *Practical Nonparametric Statistics*, 3rd edition ed. Wiley, New York, NY, USA, 1998.

[26] Cook, J. E., and Wolf, A. L. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol. 7*, 3 (1998), 215–249.

[27] Damas, C., Lambeau, B., Dupont, P., and van Lamsweerde, A. Generating annotated behavior models from end-user scenarios. *IEEE Trans. Softw. Eng. 31*, 12 (2005), 1056–1073.

[28] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[29] Deb, K. Multi-objective optimization. In *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Second Edition*. Springer, New York, NY, USA, 2014, pp. 403–449.

[30] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation 6*, 2 (2002), 182–197.

[31] Di Nardo, D., Alshahwan, N., Briand, L., and Labiche, Y. Coverage-based test case prioritisation: An industrial case study. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation (ICST)* (2013), IEEE, pp. 302–311.

[32] Do, H., and Rothermel, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering 32*, 9 (2006), 733–752.

[33] Du, M., and Li, F. Spell: Streaming parsing of system event logs. In *Proceedings of the16th IEEE International Conference on Data Mining (ICDM 2016)* (Piscataway, NJ, USA, 2016), IEEE, pp. 859–864.

[34] El-Masri, D., Petrillo, F., Guéhéneuc, Y.-G., Hamou-Lhadj, A., and Bouziane, A. A systematic literature review on automated log abstraction techniques. *Information and Software Technology 122* (2020), 106276.

[35] Elbaum, S., Chin, H. N., Dwyer, M. B., and Dokulil, J. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2006), Association for Computing Machinery, pp. 253–264.

[36] Elbaum, S., Malishevsky, A. G., and Rothermel, G. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering 28*, 2 (2002), 159–182.

[37] Emam, S. S., and Miller, J. Inferring extended probabilistic finite-state automaton models from software executions. *ACM Trans. Softw. Eng. Methodol. 27*, 1 (2018), 4:1–4:39.

[38] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program. 69*, 1 (2007), 35–45.

[39] Flemström, D., Potena, P., Sundmark, D., Afzal, W., and Bohlin, M. Similarity-based prioritization of test case automation. *Software quality journal 26*, 4 (2018), 1421–1449.

[40] Fraser, G., and Walkinshaw, N. Behaviourally adequate software testing. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST 2012)* (Piscataway, NJ, USA, 2012), IEEE, pp. 300–309.

[41] Fu, Q., Lou, J.-G., Wang, Y., and Li, J. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining (ICDM 2009)* (Los Alamitos, CA, USA, 2009), IEEE Computer Society, pp. 149–158.

[42] Fu, Q., Zhu, J., Hu, W., Lou, J.-G., Ding, R., Lin, Q., Zhang, D., and Xie, T. Where do developers log? an empirical study on logging practices in industry. In *Proceedings of the 36th International Conference*

on Software Engineering (ICSE Companion 2014) (New York, NY, USA, 2014), ACM, pp. 24–33.

[43] Gold, E. M. Language identification in the limit. *Information and Control 10*, 5 (1967), 447–474.

[44] Goldstein, M., Raz, D., and Segall, I. Experience report: Log-based behavioral differencing. In *Proceedings of the 28th International Symposium on Software Reliability Engineering (ISSRE 2017)* (Piscataway, NJ, USA, 2017), IEEE, pp. 282–293.

[45] Gopinath, R., Jensen, C., and Groce, A. Mutations: How close are they to real faults? In *Proceeding of the 25th International Symposium on Software Reliability Engineering (ISSRE)* (Los Alamitos, CA, USA, 2014), IEEE, pp. 189–200.

[46] Günther, C. W., and van der Aalst, W. M. P. Fuzzy mining-adaptive process simplification based on multi-perspective metrics. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)* (Berlin, Heidelberg, 2007), vol. 4714 of *LNCS*, Springer, pp. 328–343.

[47] Hajri, I., Goknil, A., Pastore, F., and Briand, L. C. Automating system test case classification and prioritization for use case-driven testing in product lines. *Empirical Software Engineering 25*, 5 (2020), 3711–3769.

[48] He, P., Zhu, J., He, S., Li, J., and Lyu, M. R. An evaluation study on log parsing and its use in log mining. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2016)* (Piscataway, NJ, USA, 2016), IEEE, pp. 654–661.

[49] He, P., Zhu, J., Zheng, Z., and Lyu, M. R. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of the International Conference on Web Services (ICWS 2017)* (Piscataway, NJ, USA, 2017), IEEE, pp. 33–40.

[50] Henard, C., Papadakis, M., Harman, M., Jia, Y., and Le Traon, Y. Comparing white-box and black-box test prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), IEEE, pp. 523–534.

[51] Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., and Le Traon, Y. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering 40*, 7 (2014), 650–670.

[52] Huang, Y.-C., Peng, K.-L., and Huang, C.-Y. A history-based cost-cognizant test case prioritization technique in regression testing. *J. Syst. Softw. 85*, 3 (2012), 626–637.

[53] Jia, Y., and Harman, M. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering 37*, 5 (2010), 649–678.

[54] Jia, Y., and Harman, M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering 37*, 5 (2011), 649–678.

[55] Jiang, Z. M., Hassan, A. E., Hamann, G., and Flora, P. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution: Research and Practice 20*, 4 (2008), 249–267.

[56] Jin, W., Orso, A., and Xie, T. Automated behavioral regression testing. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)* (2010), IEEE, pp. 137–146.

[57] Jorde, M., Elbaum, S., and Dwyer, M. B. Increasing test granularity by aggregating unit tests. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)* (New York, NY, USA, 2008), ACM, pp. 9–18.

[58] Jsbsim.

[59] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), Association for Computing Machinery, pp. 654–665.

[60] Kampmann, A., and Zeller, A. Carving parameterized unit tests. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (New York, NY, USA, 2019), ACM, pp. 248–249.

[61] Khare, V., Yao, X., and Deb, K. Performance scaling of multi-objective evolutionary algorithms. In *Proceedings of the 2nd International Conference on Evolutionary Multi-criterion Optimization (EMO 2003)* (Berlin, Heidelberg, 2003), vol. 2632 of *LNCS*, Springer-Verlag, pp. 376–390.

[62] Kintis, M., Papadakis, M., Papadopoulos, A., Valvis, E., Malevris, N., and Le Traon, Y. How effective are mutation testing tools? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering 23*, 4 (2018), 2426–2463.

[63] Kobayashi, S., Fukuda, K., and Esaki, H. Towards an nlp-based log template generation algorithm for system log analysis. In *Proceedings of The Ninth International Conference on Future Internet Technologies* (2014), pp. 1–4.

[64] Korel, B., Koutsogiannakis, G., and Tahat, L. H. Application of system models in regression test suite prioritization. In *Proceedings of the International Conference on Software Maintenance (ICSM)* (2008), IEEE Computer Society, pp. 247–256.

[65] Korel, B., and Laski, J. Dynamic program slicing. *Information processing letters 29*, 3 (1988), 155–163.

[66] Korel, B., Tahat, L., and Harman, M. Test prioritization using system models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)* (2005), pp. 559–568.

[67] Korel, B., Tahat, L. H., and Harman, M. Test prioritization using system models. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (2005), IEEE, pp. 559–568.

[68] Kumar, S., Khoo, S., Roychoudhury, A., and Lo, D. Mining message sequence graphs. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011)* (New York, NY, USA, 2011), ACM, pp. 91–100.

[69] Kumar, S., Khoo, S.-C., Roychoudhury, A., and Lo, D. Inferring class level specifications for distributed systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)* (Piscataway, NJ, USA, 2012), IEEE, pp. 914–924.

[70] Lang, K. J., Pearlmutter, B. A., and Price, R. A. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In *Proceedings of the 4th International Colloquium on Grammatical Inference (ICGI 1998)* (Berlin, Heidelberg, 1998), vol. 1433 of *LNCS*, Springer, pp. 1–12.

[71] Lin, H., Wang, Y., and Gong, Y. Subsuming mutation operators. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (New York, NY, USA, 2018), ACM, pp. 236–237.

[72] Liu, C., van Dongen, B., Assy, N., and van der Aalst, W. M. P. Component behavior discovery from software execution data. In *Proceedings of the Symposium Series on Computational Intelligence (SSCI 2016)* (Piscataway, NJ, USA, 2016), IEEE, pp. 1–8.

[73] Lohmann, N. Mutate++.

[74] Lou, Y., Chen, J., Zhang, L., and Hao, D. A survey on regression test-case prioritization. In *Advances in Computers*, vol. 113. Elsevier, 2019, pp. 1–46.

[75] Lou, Y., Hao, D., and Zhang, L. Mutation-based test-case prioritization in software evolution. In *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE)* (2015), IEEE, pp. 46–57.

[76] Luo, C., He, F., and Ghezzi, C. Inferring software behavioral models with mapreduce. *Sci. Comput. Program. 145* (2017), 13–36.

[77] Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), Association for Computing Machinery, pp. 643–653.

[78] Makanju, A., Zincir-Heywood, A. N., and Milios, E. E. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering 24*, 11 (2012), 1921–1936.

[79] Mariani, L., Papagiannakis, S., and Pezze, M. Compatibility and regression testing of cots-component-based software. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)* (2007), IEEE, pp. 85–95.

[80] Mariani, L., Pezzè, M., and Santoro, M. Gk-tail+ an efficient approach to learn software models. *IEEE Trans. Softw. Eng. 43*, 8 (2017), 715–738.

[81] Marinke, R., Guerra, E. M., Silveira, F. F., Azevedo, R. M., Nascimento, W., de Almeida, R. S., Demboscki, B. R., and da Silva, T. S. Towards an extensible architecture for refactoring test code. In *Proceedings of the International Conference on Computational Science and Its Applications* (Cham, Switzerland, 2019), Springer, pp. 456–471.

[82] Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., and Rothermel, G. A static approach to prioritizing junit test cases. *IEEE transactions on software engineering 38*, 6 (2012), 1258–1275.

[83] Mens, T., and Tourwé, T. A survey of software refactoring. *IEEE Transactions on software engineering 30*, 2 (2004), 126–139.

[84] Messaoudi, S., Panichella, A., Bianculli, D., Briand, L., and Sasnauskas, R. A search-based approach for accurate identification of log message formats. In *Proceedings of the 26th International Conference on Program Comprehension (ICPC 2018)* (New York, NY, USA, 2018), ACM, pp. 167–177.

[85] Messaoudi, S., Shin, D., Panichella, A., Bianculli, D., and Briand, L. Log-based slicing for system-level test cases. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (New York, NY, USA, 2021), ACM.

[86] Mi, H., Wang, H., Zhou, Y., Lyu, M. R., and Cai, H. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems 24*, 6 (2013), 1245–1255.

[87] Mills, D. L. Internet time synchronization: The network time protocol. *Transactions on Communications 39*, 10 (1991), 1482–1493.

[88] Miranda, B., Cruciani, E., Verdecchia, R., and Bertolino, A. Fast approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)* (2018), pp. 222–232.

[89] Mitchell, T. M. *Machine Learning.* McGraw-Hill, Inc., New York, NY, USA, 1997.

[90] Mizutani, M. Incremental mining of system log format. In *2013 IEEE International Conference on Services Computing* (2013), pp. 595–602.

[91] Nagaraj, K., Killian, C., and Neville, J. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI 2012)* (Berkeley, CA, USA, 2012), USENIX Association, pp. 26–26.

[92] Nagy, R., Suciu, M. A., and Dumitrescu, D. Lorenz equilibrium: Equitability in non-cooperative games. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO 2012)* (New York, NY, USA, 2012), ACM, pp. 489–496.

[93] Palomba, F., Zaidman, A., and De Lucia, A. Automatic test smell detection using information retrieval techniques. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)* (New York, NY, USA, 2018), Association for Computing Machinery, pp. 311–322.

[94] Pan, R., Bagherzadeh, M., Ghaleb, T. A., and Briand, L. Test case selection and prioritization using machine learning: A systematic literature review. *arXiv preprint arXiv:2106.13891* (2021).

[95] Panichella, A., Kifetew, F. M., and Tonella, P. Reformulating branch coverage as a many-objective optimization problem. In *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation (ICST 2015)* (Piscataway, NJ, USA, 2015), IEEE, pp. 1–10.

[96] Panichella, A., Panichella, S., Fraser, G., Sawant, A. A., and Hellendoorn, V. J. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *Proceeding of the IEEE International Conference on Software Maintenance and Evolution (ICSME)* (Los Alamitos, CA, USA, 2020), IEEE, pp. 523–533.

[97] Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings*

of the 29th Annual International Conference on Computer Science and Software Engineering (USA, 2019), IBM Corp., pp. 193–202.

[98] Petke, J., Yoo, S., Cohen, M. B., and Harman, M. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), pp. 26–36.

[99] Microsoft python-program-analysis library.

[100] Rojas, J. M., Fraser, G., and Arcuri, A. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability 26*, 5 (2016), 366–401.

[101] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering 27*, 10 (2001), 929–948.

[102] Sahni, S., and Thanvantri, V. Performance metrics: Keeping the focus on runtime. *IEEE Parallel Distributed Technology: Systems Applications 4*, 1 (1996), 43–56.

[103] Sayyad, A. S., Goseva-Popstojanova, K., Menzies, T., and Ammar, H. On parameter tuning in search based software engineering: A replicated empirical study. In *Proceedings of the 3rd International Workshop on Replication in Empirical Software Engineering Research (RESER 2013)* (Washington, DC, USA, 2013), IEEE Computer Society, pp. 84–90.

[104] Shahin, M., Babar, M. A., and Zhu, L. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access 5* (2017), 3909–3943.

[105] Shin, D., Messaoudi, S., Bianculli, D., Panichella, A., Briand, L., and Sasnauskas, R. Scalable inference of system-level models from component logs. *arXiv preprint arXiv:1908.02329* (2019).

[106] Shin, D., Yoo, S., Papadakis, M., and Bae, D.-H. Empirical evaluation of mutation-based test case prioritization techniques. *Software Testing, Verification and Reliability 29*, 1-2 (2019), e1695.

[107] Simao, A., Petrenko, A., and Maldonado, J. Comparing finite state machine test coverage criteria. *IET software 3*, 2 (2009), 91–105.

[108] Sivanandam, S., and Deepa, S. *Introduction to Genetic Algorithms.* Springer, Berlin, Heidelberg, 2008.

[109] Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., and Bacchelli, A. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution. (ICSME)* (New York, NY, USA, 2018), Association for Computing Machinery, pp. 1–12.

[110] Spieker, H., Gotlieb, A., Marijan, D., and Mossige, M. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (2017), pp. 12–22.

[111] Syswerda, G. Uniform crossover in genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms (ICGA 1989)* (San Francisco, CA, USA, 1989), Morgan Kaufmann Publishers, pp. 2–9.

[112] Tang, L., Li, T., and Perng, C.-S. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM international conference on Information and knowledge management* (2011), pp. 785–794.

[113] Tang, L., Li, T., and Perng, C.-S. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM 2011)* (New York, NY, USA, 2011), ACM, pp. 785–794.

[114] Technology, B. T. Bullseyecoverage.

[115] Thaler, S., Menkonvski, V., and Petkovic, M. Towards a neural language model for signature extraction from forensic logs. In *2017 5th International Symposium on Digital Forensic and Security (ISDFS)* (2017), pp. 1–6.

[116] Vaarandi, R. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)* (2003), Ieee, pp. 119–126.

[117] Vaarandi, R. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 3rd IEEE Workshop on IP Operations &*

Management (IPOM 2003) (Piscataway, NJ, USA, 2003), IEEE, pp. 119–126.

[118] van der Werf, J. M. E. M., van Dongen, B. F., Hurkens, C. A. J., and Serebrenik, A. Process discovery using integer linear programming. In *Proceedings of the 29th International Conference on Applications and Theory of Petri Nets (PETRI NETS 2008)* (Berlin, Heidelberg, 2008), vol. 5062 of *LNCS*, Springer, pp. 368–387.

[119] Van Deursen, A., Moonen, L., Van Den Bergh, A., and Kok, G. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP)* (New York, NY, USA, 2001), ACM, pp. 92–95.

[120] Vargha, A., and Delaney, H. D. A critique and improvement of the CL common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics 25*, 2 (2000), 101–132.

[121] Walkinshaw, N., Bogdanov, K., Damas, C., Lambeau, B., and Dupont, P. A framework for the competitive evaluation of model inference techniques. In *Proceedings of the First International Workshop on Model Inference In Testing (MIIT 2010)* (New York, NY, USA, 2010), ACM, pp. 1–9.

[122] Walkinshaw, N., Lambeau, B., Damas, C., Bogdanov, K., and Dupont, P. Stamina: A competition to encourage the development and assessment of software model inference techniques. *Empir. Softw. Eng. 18*, 4 (Aug 2013), 791–824.

[123] Walkinshaw, N., Taylor, R., and Derrick, J. Inferring extended finite state machine models from software executions. *Empir. Softw. Eng. 21*, 3 (2016), 811–853.

[124] Wang, S., Ali, S., Yue, T., Li, Y., and Liaaen, M. A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)* (New York, NY, USA, 2016), ACM, pp. 631–642.

[125] Wang, S., Lo, D., Jiang, L., Maoz, S., and Budi, A. Scalable parallelization of specification mining using distributed computing. In *The Art and Science of Analyzing Software Data*. Morgan Kaufmann, 2015, pp. 623–648.

[126] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering SE-10*, 4 (1984), 352–357.

[127] Whaley, J., Martin, M. C., and Lam, M. S. Automatic extraction of object-oriented component interfaces. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)* (New York, NY, USA, 2002), ACM, pp. 218–228.

[128] Wong, W. E., Debroy, V., Golden, R., Xu, X., and Thuraisingham, B. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability 61*, 1 (2012), 149–169.

[129] Xie, T., and Notkin, D. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on software Engineering 31*, 10 (2005), 869–883.

[130] Xu, W. *System Problem Detection by Mining Console Logs.* PhD thesis, University of California Berkeley, 2010.

[131] Xuan, J., and Monperrus, M. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)* (New York, NY, USA, 2014), ACM, pp. 52–63.

[132] Yoo, S., Binkley, D. W., and Eastman, R. D. Observational slicing based on visual semantics. *J. Syst. Softw. 129* (2017), 60–78.

[133] Yoo, S., and Harman, M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability 22*, 2 (2012), 67–120.

[134] Yoo, S., and Harman, M. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability 22*, 2 (2012), 67–120.

[135] Yuan, D., Park, S., and Zhou, Y. Characterizing logging practices in open-source software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)* (Piscataway, NJ, USA, 2012), IEEE, pp. 102–112.

[136] Zhang, J., Zhang, L., Harman, M., Hao, D., Jia, Y., and Zhang, L. Predictive mutation testing. *IEEE Transactions on Software Engineering 45*, 9 (2018), 898–918.

[137] Zhou, Z. Q., Sinaga, A., Susilo, W., Zhao, L., and Cai, K.-Y. A cost-effective software testing strategy employing online feedback information. *Inf. Sci. 422*, C (2018), 318–335.

[138] Zhu, J., He, P., Fu, Q., Zhang, H., Lyu, M. R., and Zhang, D. Learning to log: Helping developers make informed logging decisions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)* (Piscataway, NJ, USA, 2015), IEEE, pp. 415–425.