technische universität
dortmund

Masterarbeit

Laufzeitüberwachung eingebetteter Systeme zur Detektion von Softwarefehlern mit AspectC++

Software-Fault Detection in Embedded Systems by Runtime Monitoring with AspectC++

Simon Dierl

21. August 2018

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
M. Sc. Ulrich Thomas Gabor

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Eingebettete Systemsoftware
`https://ess.cs.tu-dortmund.de`

Abstract

In the development and maintenance of software, the localization of the root cause of bugs is challenging and time-consuming. Spectrum-based fault localization automates identification of root causes by analyzing a set of test runs. Execution of the runs has to be triggered manually.

To automate the process, this thesis employs automation to gather relevant information from a single continuous execution. The automation is extended to operate without domain knowledge by inferring models from well-behaving software and detect deviant behavior by runtime model checking. To this purpose, driver interaction models are introduced to describe the interactive behavior of drivers and peripheral components.

Empirical examination demonstrated the feasibility of the approach. Driver interaction models were able to detect over 50 % of the introduced defects and spectrum-based fault localization was subsequently able to identify the defect's location with near-perfect accuracy. Domain-agnostic approaches appear to be a useful extension to spectrum-based fault localization, while the technique itself is applicable in embedded scenarios. The applications of driver interaction models warrant further study.

Contents

Chapter 1

Introduction

Since the first program was written, bugs have been a steady companion to software developers. In her notes accompanying a 1842 translation of Menabrea's paper on the subject, Ada, Countess of Lovelace, famously proposed a set of computations that could be run by Charles Babbage's analytical engine, a mechanical, Turing-complete computer that only existed as a set of sketches [Men42]. Since her work predates the first operational computer by about a century, she is regarded as the first computer programmer.

In her Note G, a program to compute Bernoulli numbers is described. The program is reproduced in Figure 1.1. In the "Statement of Results" column, it can be seen that step 4 is supposed to compute $\frac{2n-1}{2n+1}$. However, the operation in the third column interchanges the source registers, accidentally computing $\frac{2n+1}{2n-1}$.

After computers came into widespread use, a plethora of approaches to avoid the introduction of bugs or reduce their severity were tried, but either fell short of their goal or have not yet gained widespread adoption. In the meantime, an immense corpus of software has been developed that most certainly *does* contain bugs. Newer software builds on older solutions, relying on their correctness. This amplifies the impact of bugs in these foundations.

While the idea of rewriting all legacy software, "doing it right this time", might be appealing to debug-weary software developers and users worrying about the safety of their data, it is most definitely not feasible. However, it might be possible to reduce the amount of bugs in existing software to achieve the same goal.

Techniques to identify and eliminate bugs already exist. Commonly, debuggers are used during the development of software to observe the execution of programs, attempting to discover bugs in the process. The discovery of bugs during production use is often aided by the creation of log files that might reveal both an error and its cause during later analysis.

Unfortunately, these approaches consume scarce developer time. A human either has to operate a debugger or interpret log files, and then delve into source code to

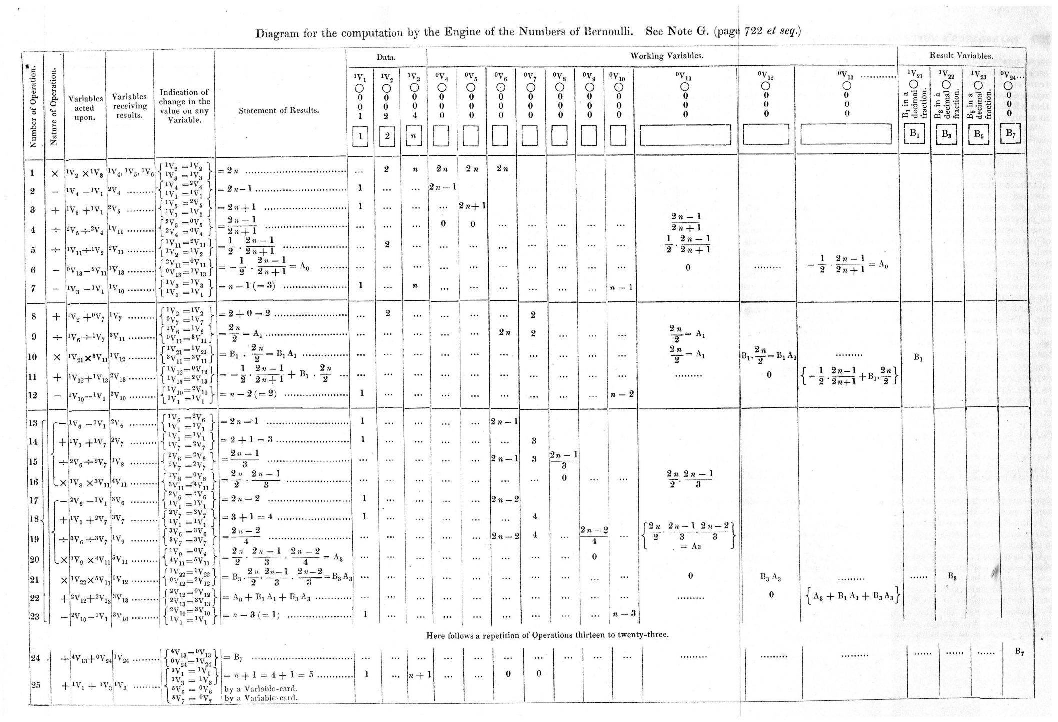


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data.			Working Variables.										Result Variables.			
						$^1V_1$ ○ 0 0 0 1 $\boxed{1}$	$^1V_2$ ○ 0 0 0 2 $\boxed{2}$	$^1V_3$ ○ 0 0 0 4 $\boxed{n}$	$^0V_4$ ○ 0 0 0 0	$^0V_5$ ○ 0 0 0 0	$^0V_6$ ○ 0 0 0 0	$^0V_7$ ○ 0 0 0 0	$^0V_8$ ○ 0 0 0 0	$^0V_9$ ○ 0 0 0 0	$^0V_{10}$ ○ 0 0 0 0	$^0V_{11}$ ○ 0 0 0	$^0V_{12}$ ○ 0 0 0	$^0V_{13}$ ○ 0 0 0	$^1V_{21}$ ○ B_1 in a decimal fraction. $\boxed{B_1}$	$^1V_{22}$ ○ B_3 in a decimal fraction. $\boxed{B_3}$	$^1V_{23}$ ○ B_5 in a decimal fraction. $\boxed{B_5}$	$^0V_{24}$... ○ 0 0 0 $\boxed{B_7}$
1	×	$^1V_2 \times {}^1V_3$	$^1V_4, {}^1V_5, {}^1V_6$	$\{{}^1V_2 = {}^1V_2;\ {}^1V_3 = {}^1V_3\}$	$= 2n$	...	2	n	$2n$	$2n$	$2n$											
2	−	$^1V_4 - {}^1V_1$	$^2V_4$	$\{{}^1V_4 = {}^2V_4;\ {}^1V_1 = {}^1V_1\}$	$= 2n-1$	1	...	...	$2n-1$													
3	+	$^1V_5 + {}^1V_1$	$^2V_5$	$\{{}^1V_5 = {}^2V_5;\ {}^1V_1 = {}^1V_1\}$	$= 2n+1$	1	...	...	...	$2n+1$												
4	÷	$^2V_5 \div {}^2V_4$	$^1V_{11}$	$\{{}^2V_5 = {}^0V_5;\ {}^2V_4 = {}^0V_4\}$	$= \frac{2n-1}{2n+1}$	...	...	...	0	0	...	...	...	...	...	$\frac{2n-1}{2n+1}$						
5	÷	$^1V_{11} \div {}^1V_2$	$^2V_{11}$	$\{{}^1V_{11} = {}^2V_{11};\ {}^1V_2 = {}^1V_2\}$	$= \frac{1}{2} \cdot \frac{2n-1}{2n+1}$	...	2	...	...	...	...	...	...	...	...	$\frac{1}{2} \cdot \frac{2n-1}{2n+1}$						
6	−	$^0V_{13} - {}^2V_{11}$	$^1V_{13}$	$\{{}^2V_{11} = {}^0V_{11};\ {}^0V_{13} = {}^1V_{13}\}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$	...	...	...	...	...	...	...	...	...	...	0	....	$-\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$				
7	−	$^1V_3 - {}^1V_1$	$^1V_{10}$	$\{{}^1V_3 = {}^1V_3;\ {}^1V_1 = {}^1V_1\}$	$= n - 1 \ (= 3)$	1	...	n	...	...	...	...	...	...	$n-1$							
8	+	$^1V_2 + {}^0V_7$	$^1V_7$	$\{{}^1V_2 = {}^1V_2;\ {}^0V_7 = {}^1V_7\}$	$= 2 + 0 = 2$	...	2	...	...	...	...	2										
9	÷	$^1V_6 \div {}^1V_7$	$^3V_{11}$	$\{{}^1V_6 = {}^1V_6;\ {}^0V_{11} = {}^3V_{11}\}$	$= \frac{2n}{2} = A_1$	...	...	...	...	...	$2n$	2	...	...	...	$\frac{2n}{2} = A_1$						
10	×	$^1V_{21} \times {}^3V_{11}$	$^1V_{12}$	$\{{}^1V_{21} = {}^1V_{21};\ {}^3V_{11} = {}^3V_{11}\}$	$= B_1 \cdot \frac{2n}{2} = B_1 A_1$	...	...	...	...	...	...	...	...	...	...	$\frac{2n}{2} = A_1$	$B_1 \cdot \frac{2n}{2} = B_1 A_1$	....	B_1			
11	+	$^1V_{12} + {}^1V_{13}$	$^2V_{13}$	$\{{}^1V_{12} = {}^0V_{12};\ {}^1V_{13} = {}^2V_{13}\}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}$ ...	...	...	...	...	...	...	...	...	...	...	....	0	$\left\{-\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}\right\}$				
12	−	$^1V_{10} - {}^1V_1$	$^2V_{10}$	$\{{}^1V_{10} = {}^2V_{10};\ {}^1V_1 = {}^1V_1\}$	$= n - 2 \ (= 2)$	1	...	...	...	...	...	...	...	...	$n-2$							
13	−	$^1V_6 - {}^1V_1$	$^2V_6$	$\{{}^1V_6 = {}^2V_6;\ {}^1V_1 = {}^1V_1\}$	$= 2n - 1$	1	...	...	...	...	$2n-1$											
14	+	$^1V_1 + {}^1V_7$	$^2V_7$	$\{{}^1V_1 = {}^1V_1;\ {}^1V_7 = {}^2V_7\}$	$= 2 + 1 = 3$	1	...	...	...	...	...	3										
15	÷	$^2V_6 \div {}^2V_7$	$^1V_8$	$\{{}^2V_6 = {}^2V_6;\ {}^2V_7 = {}^2V_7\}$	$= \frac{2n-1}{3}$	...	...	...	...	...	$2n-1$	3	$\frac{2n-1}{3}$									
16	×	$^1V_8 \times {}^3V_{11}$	$^4V_{11}$	$\{{}^1V_8 = {}^0V_8;\ {}^3V_{11} = {}^4V_{11}\}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3}$	...	...	...	...	...	...	...	0	...	...	$\frac{2n}{2} \cdot \frac{2n-1}{3}$						
17	−	$^2V_6 - {}^1V_1$	$^3V_6$	$\{{}^2V_6 = {}^3V_6;\ {}^1V_1 = {}^1V_1\}$	$= 2n - 2$	1	...	...	...	...	$2n-2$											
18	+	$^1V_1 + {}^2V_7$	$^3V_7$	$\{{}^2V_7 = {}^3V_7;\ {}^1V_1 = {}^1V_1\}$	$= 3 + 1 = 4$	1	...	...	...	...	...	4										
19	÷	$^3V_6 \div {}^3V_7$	$^1V_9$	$\{{}^3V_6 = {}^3V_6;\ {}^3V_7 = {}^3V_7\}$	$= \frac{2n-2}{4}$	...	...	...	...	...	$2n-2$	4	...	$\frac{2n-2}{4}$	...	$\left\{\frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3} = A_3\right\}$						
20	×	$^1V_9 \times {}^4V_{11}$	$^5V_{11}$	$\{{}^1V_9 = {}^0V_9;\ {}^4V_{11} = {}^5V_{11}\}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{4} = A_3$	...	...	...	...	...	...	...	...	0								
21	×	$^1V_{22} \times {}^5V_{11}$	$^0V_{12}$	$\{{}^1V_{22} = {}^1V_{22};\ {}^0V_{12} = {}^2V_{12}\}$	$= B_3 \cdot \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{3} = B_3 A_3$	...	...	...	...	...	...	...	...	...	...	0	$B_3 A_3$	....	......	B_3		
22	+	$^2V_{12} + {}^2V_{13}$	$^3V_{13}$	$\{{}^2V_{12} = {}^0V_{12};\ {}^2V_{13} = {}^3V_{13}\}$	$= A_0 + B_1 A_1 + B_3 A_3$	...	...	...	...	...	...	...	...	...	...	....	0	$\{A_3 + B_1 A_1 + B_3 A_3\}$				
23	−	$^2V_{10} - {}^1V_1$	$^3V_{10}$	$\{{}^2V_{10} = {}^3V_{10};\ {}^1V_1 = {}^1V_1\}$	$= n - 3 \ (= 1)$	1	...	...	...	...	...	...	...	...	$n-3$							
						Here follows a repetition of Operations thirteen to twenty-three.																
24	+	$^4V_{13} + {}^0V_{24}$	$^1V_{24}$	$\{{}^4V_{13} = {}^0V_{13};\ {}^0V_{24} = {}^1V_{24}\}$	$= B_7$	...	...	...	...	...	...	...	...	...	...	....	....	....	......	......	......	B_7
25	+	$^1V_1 + {}^1V_3$	$^1V_3$	$\{{}^1V_1 = {}^1V_1;\ {}^1V_3 = {}^1V_3;\ {}^5V_6 = {}^0V_6;\ {}^5V_7 = {}^0V_7\}$	$= n + 1 = 4 + 1 = 5$ .... by a Variable-card. by a Variable-card.	1	...	$n+1$	...	...	0	0										

Figure 1.1: Ada, Countess of Lovelace's program to compute Bernoulli numbers[1]

locate a bug. Additional roadblocks to debugging might be hard-to-read legacy code, unusual coding conventions and lack of architectural knowledge.

Even without these obstacles, finding the bug is hardly guaranteed. A log file might omit relevant information due to storage constraints. If the bug only occurs in a specific environment, e.g., due to being triggered by a specific combination of input data, debugging can only be performed in a faithful emulation of this setting. Infrequently triggering bugs would require debugging sessions over a time span of hours or even days. In the worst case, the presence of a debugger or detailed logging might even suppress the bug.

Such hurdles to debugging and logging are more common in low-level software, e.g., operating systems and software running on embedded devices, although they are hardly exclusive to it. However, most approaches suited to embedded systems can scale *up* to more forgiving environments.

This thesis presents a different approach to localizing bugs that can meet those challenges. By automating monitoring of a system over long periods and automatically suggesting likely causes of a bug, manual work can be reduced and developer time saved.

[1]Digitization from The Erwin Tomash Library On The History Of Computing, `https://www.computer.org/cms/Tomash catalog web/index.htm`.

1.1 Objectives

A framework for the localization of bugs that implemented this approache was developed for this thesis. It is capable of monitoring a system during its run time, collecting information about it and creating a report from the processed data that can be displayed to a human developer. To guide implementation and development, four major objectives were defined, outlining the requirements the framework must satisfy.

The first requirement is that the framework should **assist in the localization of bugs**. By providing accurate suggestions on locations in a program's source code where a bug might originate, developers can focus their efforts on these locations, skipping unrelated components. This objective was achieved using spectrum-based fault localization. Some extensions to the process are investigated for the thesis.

Classic spectrum-based fault localization is not suited to localize bugs in long-running system due to requiring occasional manual intervention. To save valuable developer time, analysis should instead be performed **autonomous**ly over an extended time period; human labor should only be required to analyze the results, not gather them. Autonomy was achieved using generic analysis components, oracles and transaction detectors, that were created using machine learning.

Since the framework explicitly targets existing and even legacy software, the analysis framework must be **retrofittable** into preexisting software with little effort. Ideally, the software under examination should require no modification, in practice, small and localized changes might be unavoidable. Since certain bugs might be obscured by invasive diagnostics, this gathering of information must be as **non-intrusive** as possible. This requirement was satisified using aspect-oriented programming techniques.

In privileged operating system code, a bug can easily corrupt unrelated data or require a hardware reset to restore functionality. To facilitate analysis of such bugs, the analyzing system must **operate in isolation** from the diagnosed system to maintain integrity of the stored information. Since this isolation can not be guaranteed on the same system, **physical isolation** – separation of examined and analyzing system – was necessary.

1.2 Contributions

Three novel contributions to the state of the art are presented in this thesis:

1. The extensions to spectrum-based fault localization, with the exception of method call sequence hit spectra, have not previously been discussed.

2. The use of generic, machine-learned oracles and transaction detectors is a new approach. The driver interaction models used as generic oracles are a novel method for modeling the communications between a driver and the managed periphery.
3. An autonomously operating fault localization framework using embedded devices has not yet been investigated; past approaches were limited to fault localization with major human involvement.

1.3 Structure of this Work

This thesis is structured into nine chapters. Chapter 2 describes the use of spectrum-based fault localization to pinpoint bugs. These ideas are extended to long-running systems in Chapter 3. Chapter 4 then outlines how analysis data can be acquired using aspect-oriented programming.

These foundations having been laid, Chapter 5 describes the implementation of the analysis framework using these concepts; Chapter 6 then outlines the integration into the operating system CyPhOS and the issues encountered.

Chapter 7 details the experiments used to evaluate the work and outlines the results, followed by a discussion on related work that was not adapted for this thesis in Chapter 8. Finally, in Chapter 9, conclusions from these results are drawn and an outlook on future work is given.

Chapter 2

Spectrum-Based Fault Localization

This chapter discusses the use of spectrum-based fault localization to the localize bugs. First, a standard terminology to formalize the notion of "bugs" will be introduced, then, the fundamentals of spectrum-based fault localization and how it was applied in the scope of this thesis are described. Finally, three extensions to the process are presented.

2.1 Standard Terminology for Defects

Avižienis et al. provide a standard terminology to formally describe "buggy software". Using this terminology, a **system** [Avi+04] is an entity (e.g., a program) that interacts with other entities. These other entities might be other systems, human users, hardware etc. A system can often be broken up into smaller systems, **components** [Avi+04].

The intended operation, including functionality, performance, etc., of a system is called its **function** [Avi+04]. In contrast to this, a system's **behavior** [Avi+04] is its actual operation, i.e., the implementation of the desired function. The part of the behavior that can be perceived by a system's users is called **service** [Avi+04].

As an example, assume that an array-backed list class for use in a collection library is being implemented. In this case:

- the system is the list implementation,
- the list's function comprises the *intended* semantics of the list's operations, e.g., adding and removing of elements, but also the storage implementation hidden from a user,
- the list's behavior comprises the semantics "as programmed", including the implementation of its operations, but also routines for on-demand storage enlargement, and

- the service comprises the semantics “as programmed”, *except* for the internal state.

Having formalized a concept of “software” and its intended operation, the notion of “bugs” can be replaced by more precise terms separating cause and effect. A **failure** [Avi+04] occurs when a system’s service deviates from the service intended in its function. Meanwhile, a **fault** [Avi+04] is the cause of a failure. **Fault localization** [Avi+04] attempts to pinpoint the root cause of an observed failure, i.e., find the underlying fault.

To continue the previous example, assume that the list implementation erroneously ignores removal operations. Then,

- the fault is located in the code that is supposed to handle removals, but
- the failure only manifests when a client recovers an element from the list that had previously been deleted, and
- fault localization would then try to locate a function or even line of code in the removal code that causes this failure.

While this terminology is applicable to a many types of system, this thesis concentrates on **programs** that can be separated into components such as modules, functions, lines of code or statements.

For added clarity, a program’s execution will be referred to as a **run** and the program as **running**. A run begins with the program’s start and ends with its termination, if at all. The term **execution** shall be reserved for components. During a program’s run, some or all of its components are **executed** one or more times.

If a program’s run is recorded, the chronological sequence of all component’s executions[1] is called the run’s **trace** [DLZ05]. A trace of the sequential executions of the components A, B and C is written as

$$\langle A, B, C \rangle.$$

As an additional convention, if a failure is observed during a program’s run, this run is called **failing**, otherwise **succeeding**. These terms also apply to a corresponding trace. Components that contain a fault are called **faulty**.

This terminology allows for a precise description of a “bug”, its cause, effect and location.

[1]Considerations for multi-threaded programs will be discussed in Section 2.6.

2.2 Basic Concepts

Spectrum-based fault localization (SBFL) exploits the fact that if a failure was observed during a program's run, at least one faulty component must have been executed and must therefore be present in the run's trace.

A single trace only permits the conclusion that components that were not executed can not have contributed to the failure. The combination of data from multiple failing and succeeding runs allows to pinpoint the faulty component further. SBFL assumes that, when examining a set of traces, the execution of a faulty component is correlated with the observation of a failure.

To study this correlation, the components executed during a run need to be recorded. Given a program separated into k components $C_0, \ldots, C_{k-1}$ that was run a single time, a vector $S \in \{0,1\}^k$

$$S_i = 1 \iff C_i \text{ was executed}$$

that records which components were executed *at least* once is called the run's **spectrum** [Rep+97]. Next, a recording of the failures is needed. If the program was run ℓ times, a vector $e \in \{0,1\}^\ell$ such that

$$e^j = 1 \iff \text{run } j \text{ is failing}$$

is called the **error vector** [AZG08].

To visualize the correlation, spectra and the error vector can be written side by side to form an **observation matrix** [AZG08]. Let $S_i^j \in \{0,1\}$ denote if C_i was executed in the j-th run. Using the above notation, the matrix

$$O = \left(\begin{array}{cccc|c} S_0^0 & S_1^0 & \ldots & S_{k-1}^0 & f^0 \\ S_0^1 & S_1^1 & \ldots & S_{k-1}^1 & f^1 \\ \vdots & \ddots & & \vdots & \vdots \\ \vdots & & \ddots & \vdots & \vdots \\ S_0^{\ell-1} & S_2^{\ell-1} & \ldots & S_{k-1}^{\ell-1} & f^{\ell-1} \end{array}\right)$$

would be the observation matrix of the running example. A *row* shows a single run's spectrum, while a *column* records a single component's behavior over multiple runs.

Since SBFL tries to identify a faulty component by high correlation of its execution with failed runs, the columns in this matrix need to be compared to the error vector. The i-th column of the observation matrix (a vector $c^i \in \{0,1\}^\ell$) will be called the i-th **component vector**.

$$m_{\text{Barinel}} = 1 - \frac{A_{1,0}}{S+1} \qquad \text{(Barinel [AZG09])}$$

$$m_{\text{D}^*} = \frac{(A_{1,1})^*}{A_{1,0} + (F - A_{1,1})} \qquad \text{(D}^* \text{ [Won+14])}$$

$$m_{\text{Ochiai}} = \frac{A_{1,1}}{\sqrt{F \cdot (A_{1,1} + A_{1,0})}} \qquad \text{(Ochiai [Och57])}$$

$$m_{\text{Op2}} = A_{1,1} - \frac{A_{1,0}}{S+1} \qquad \text{(Op2 [NLR11])}$$

$$m_{\text{Tarantula}} = \frac{\frac{A_{1,1}}{F}}{\frac{A_{1,1}}{F} + \frac{A_{1,0}}{S}} \qquad \text{(Tarantula [JH05])}$$

Figure 2.1: Implemented suspiciousness metrics

Given a large set of observations, fault localization can be performed by measuring the similarity between each component vector and the error vector and ranking all components by similarity. The faulty component should then be assigned a high rank. Next, a developer would use the ranking to examine higher-ranked components first, quickly finding the high-ranked faulty component.

2.3 Suspiciousness Metrics

After performing data acquisition, the measurement of similarity between a component's execution and the failed runs is used to derive the ranking of components. This similarity is measured by a **suspiciousness metric** [Xie+13], a function that maps a component vector and an error vector to a similarity score, with more similar vectors resulting in greater score. This score is called the component's **suspiciousness** [JH05]; components can be ranked by it.

Most suspiciousness metrics do not examine every element of both input vectors. Instead, for input vectors S_i, f, they only use the counts for the four different combinations of vector elements (component missing in succeeding run, component present in succeeding run, component missing in failing run, component present in failing run).

This allows for an extremely compact storage of information that scales logarithmically in the number of traces. Given the running example and a component i,

```
Input: A program $P$ with components $C_0, \dots, C_{k-1}$
Input: A suspiciousness metric using only aggregated data $m$
Input: Iteration count $R$
1 begin
2     Initialize aggregate matrices $A$ for every component with entries $\coloneqq 0$;
3     for $i \in \{0, \dots, R\}$ do
4         Run $P$, record a spectrum $S_.$;
5         if a failure occurred then $f \coloneqq 1$ else $f \coloneqq 0$;
6         for $j \in \{0, \dots, k-1\}$ do $A^j_{S_i,f} \coloneqq A^j_{S_i,f} + 1$;
7     for $C \in \{C_0, \dots, C_{k-1}\}$ do compute the suspiciousness of $C$ using $m$;
8     $C_{s(0)}, \dots, C_{s(k-1)} \coloneqq$ the components, sorted by suspiciousness;
9     for $C \in \{C_{s(0)}, \dots, C_{s(k-1)}\}$ do manually check the component for faults;
```

Listing 2.1: The SBFL workflow [Abr+09]

the matrix $A^i \in \mathbb{Z}^{2\times 2}_{\geq 0}$ such that

$$A^i = \begin{pmatrix} |\{j \mid S^j_i = 0, f^j = 0\}| & |\{j \mid S^j_i = 0, f^j = 1\}| \\ |\{j \mid S^j_i = 1, f^j = 0\}| & |\{j \mid S^j_i = 1, f^j = 1\}| \end{pmatrix}$$

is called an **aggregate matrix** [Abr+09]. Note that

$$A^i_{m,n} = |\{j \mid S^j_i = m, f^j = n\}|.$$

The exact meaning of "more similar" is not defined in a general way, but individually by metrics. The suspiciousness metrics that were implemented for this thesis are shown in Figure 2.1, using the following shorthand: for all metrics, let A be the aggregate matrix for the examined component and let

$$F = A_{0,1} + A_{1,1} \qquad S = A_{0,0} + A_{1,0}$$

be the total failure and success counts.

The D* suspiciousness metric is parameterized by supplying a custom exponent $*$. To keep the scores in a reasonable range, only $* = 2$ and $* = 3$ were examined. In degenerated cases, some metrics might not be well-defined due to divides by zero. In these cases, the implementation defaults to a sensible value, e.g., zero if the component was never involved in a failing run and $A^*_{1,1}$ if the component was never involved in a succeeding run for D*.

Combining the two building blocks yields the SBFL workflow for fault localization outlined in Listing 2.1. The program is run multiple times and for every component, an aggregate matrix is created. Then, a suspiciousness score is computed for every component; these are then ordered according to the score.

If failures persist, more iterations can be applied to discover a new fault in every iteration [JHS02]. This repetition necessitates that the program can easily be run multiple times and detection of failures is simple.

2.4 Method Call Sequence Hit Spectra

The first extension to the process is geared towards the use of methods as components when performing fault localization for programs written in object-oriented languages and was originally proposed by Dallmeier, Lindig, and Zeller for Java programs. When implementing an SBFL-based approach, they employed the aspect weaver AspectJ[2] to instrument the target program.

Since AspectJ can only instrument methods, not statements, the former were the only type of component available for SBFL. Generating a spectrum using the program's methods as components would result in very coarse data. To improve upon this, **method call sequences** were examined instead. Instead of only recording called functions, method call sequences are tuples containing the last k calls in addition to the called function.

These sequences are then used as components to generate spectra [DLZ05]. The resulting spectra were named **method call sequence hit spectra (MCSHS)** by Wong et al. [Won+16]. For example, assume that during a run, the following trace of method executions would be encountered:

$$\langle f(), g(), h(), g(), f() \rangle.$$

Using methods as components, the following components would be recorded in the spectrum:

$$\{f(), g(), h()\}.$$

In a method call sequence hit spectrum with a sequence length of $k = 2$, the set of components

$$\{(f(), g()), (g(), h()), (h(), g()), (g(), f())\}$$

would be marked in the spectrum instead. The sudden appearance of $(f(), f())$ in a failing run's spectrum could then be flagged as suspicious, while the appearance of $f()$ in the simple spectrum would not.

This approach yields suspiciousness scores for sequences. To obtain a score for a single method, an average over the scores of all sequences containing the method is computed.

[2]`https://www.eclipse.org/aspectj/`

$$f_{\text{integral}}(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \\ 2 & \text{if } x < 0 \end{cases} \qquad f_{\text{floating point}}(x) = \begin{cases} 0 & \text{if } x \equiv +0 \\ 1 & \text{if } x \equiv -0 \\ 2 & \text{if } x > 0 \\ 3 & \text{if } x < 0 \\ 4 & \text{if } x \equiv +\infty \\ 5 & \text{if } x \equiv -\infty \\ 6 & \text{if } x \equiv \texttt{NaN} \end{cases}$$

$$f_{\text{boolean}}(x) = \begin{cases} 0 & \text{if } x = \texttt{false} \\ 1 & \text{if } x = \texttt{true} \end{cases} \qquad f_{\text{pointer}}(x) = \begin{cases} 0 & \text{if } x = \texttt{nullptr} \\ 1 & \text{if } x \neq \texttt{nullptr} \end{cases}$$

Figure 2.2: All implemented classifiers

Dallmeier, Lindig, and Zeller noted that suspiciousness metrics are generally not well suited for diagnosing faults of omission, i.e., elements *missing* from a spectrum. Examples for omissions constituting a fault are a forgotten closure of a file handles or the release of a lock. To mitigate this shortcoming, a custom suspiciousness metric was devised, penalizing both abnormally high and low correlation of a component vector with the error vector in an attempt to identify missing method call sequences.

This thesis implements MCSHS with a configurable sequence length. A sequence length of one is equivalent to disabling the feature. The custom metric described by Dallmeier, Lindig, and Zeller was not implemented.

An empirical study using defects mined from two open-source software projects, NanoXML and AspectJ, found that sequence lengths between 3 and 5 resulted in promising fault localization accuracy. To facilitate use in debugging, the Eclipse plugin AMPLE[3] was developed that automatically attempts to localize faults that caused JUnit tests to fail [DLZ05].

2.5 Parameter Classification

The second extension is also designed to enable the use of functions and methods as components by examining the parameters passed to them. The behavior of a function often depends on its parameters. For example, passing a `nullptr` to a

[3]`https://www.st.cs.uni-saarland.de/ample/`

C++ function might cause an out-of-bounds memory access and cause a failure, while the function would perform as expected for any other input. Recording parameters can result in deeper insight when using method call sequences, as a sequence might show a parameter being passed between functions.

Transmitting the in-memory representation of every parameter in the examined program to the analyzer is not feasible; the parameters might require excessive storage or can be dependent on the examined program's state (e.g., a pointer). However, for some parameter types, **classification** can be applied by examining the parameter and categorizing it according to characteristics of the original type.

Classification of values is performed by a **classifier**, a function that maps all values belonging to a type to a limited numerical range. For the thesis, the range $[0, 6]$ was chosen, although larger ranges might be necessary for more complex types. Effectively, the classified type was separated into 7 or less equivalence classes. For a classified type, the values in the classifier's image are called **representatives**.

For the analysis framework, the four classifiers shown in Figure 2.2 were developed to handle most common C++ types. Integral types were classified using the signum function; booleans and pointers were classified as 1 or 0 according to their truth value. Floating point numbers were classified using their signum for common values, while negative zero, infinity and `NaN` were represented with separate values.

2.6 Thread-Aware Analysis

The third extension facilitates analysis of multi-threaded systems in two different ways. To analyze both empirically, the feature was made configurable in the analysis framework.

Trace data can – independent of the originating thread of execution – be analyzed in chronological order, i.e., the sequence of events as they were recorded. This should uncover faults such as race conditions, especially when using MCSHS. However, information about the actual flow of the program might be lost and non-faulty components running in parallel to faulty ones will accumulate suspiciousness.

The alternative to that approach is separation of the concurrent threads of execution, so-called **thread separation**. Each thread will be recorded into a separate spectrum and the data is analyzed in isolation. This increases storage and CPU load and might not reliably diagnose failures from multiple thread's interactions.

Chapter 3

SBFL in Long-Running Systems

The traditional approach to spectrum-based fault localization presented in the previous chapter is not suitable for every type of software. A restart may be costly (e.g., a server in a production environment) or a single run exercises too many components (e.g., an operating system during boot) to leave a useful spectrum. These issues are endemic to long-running systems.

This chapter discusses an extension to SBFL that resolves such issues by implementing automated analysis components, oracles and transaction detectors. Next, three approaches to derive these components without domain knowledge are presented. Finally, another extension to the SBFL process itself is discussed.

3.1 Oracles and Transaction Detectors

Oracles and transaction detectors were introduced by Casanova et al. to provide self-diagnosis mechanisms for long-running server clusters in productive use. They proposed a mechanism to separate a single long run and its trace into several smaller sub-traces. These are then tested for failures. The non-overlapping, continuous and finite sub-traces are known as **transactions** [Cas+11]; every element of the trace belongs to exactly one transaction. A transaction again constitutes a trace.

A function that splits a trace into transactions is called a **transaction detector** [Cas+11]. Two transaction detectors can be combined by pooling the demarcations between transactions found by both detectors. Functions responsible for detecting failures in a trace are called **oracles** [Cas+11]. By using transactions as runs in an otherwise classic SBFL approach, analysis of long-running systems becomes feasible. This workflow is illustrated in Figure 3.1.

As an example for transactions and oracles, a web server answering a single request is suggested as a transaction; an oracle would examine the reply and flag a late or missing reply as a failure. Neither the oracle nor the transaction detector needs

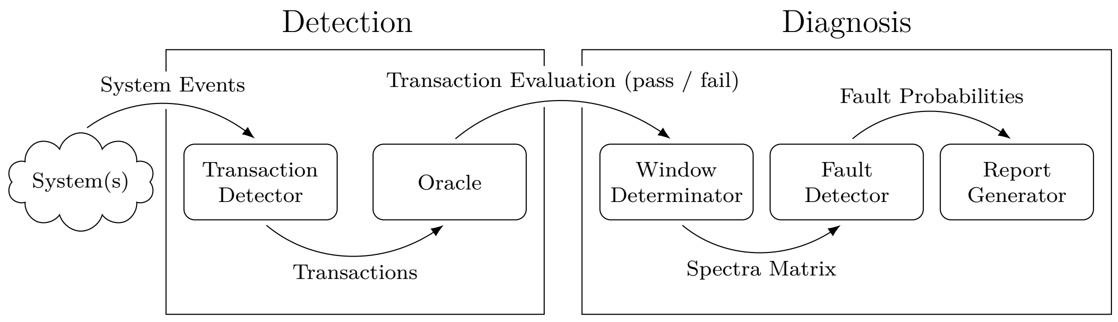


Figure 3.1: The experiment framework used by Casanova et al. (illustration from [Cas+11])

to be aware of the server's implementation details, only of the protocol it uses to communicate with other services. This system could scale to multiple transaction types, such as clients communicating with a load balancer, the load balancer forwarding requests to a web server, which in turn would use SQL to query one or more database servers.

This thesis uses transactions to extract SBFL data from a long-running system, however, its focus is on resource-constrained, embedded systems. Instead of domain-aware oracles and transaction detectors, generic components were derived from behavioral properties of a non-failing program using machine learning techniques. The resulting components were therefore created without requiring knowledge about the program's function.

3.2 Software Behavior Graphs

The first generic oracle created for the framework observes the program's function and method calls. When running the program, invocations of its functions and methods can be observed, although some caller-callee-pairs will not be encountered during normal operation. The sudden appearance of a previously unseen call can indicate a fault.

The observed calls can be stored in a **software behavior graph (SBG)** [Liu+05]

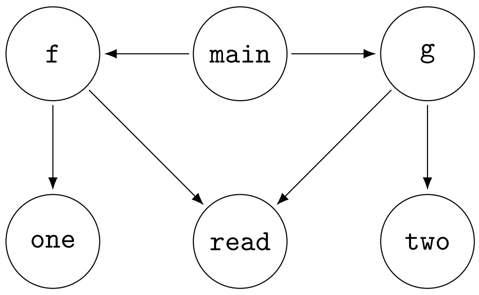
$$G = (V, E),$$

where the nodes V are the functions and methods of the program and an edge (s, t) is contained in E if and only if a call from s to t was observed.

Software behavior graphs were introduced by Liu et al. for fault localization. In their corresponding publication, the analysis was implemented using data mining on the graph itself. Since this requires more computational power than an embedded

```c
int read();
int one() { return 1; }
int two() { return 2; }
int f() {
    return read() ? one() : two();
}
int g() {
    return read() ? two() : one();
}
int main() { f(); g(); }
```

(a) program


$$\begin{tikzcd}
f \arrow[d] \arrow[dr] & main \arrow[l] \arrow[r] & g \arrow[dl] \arrow[d] \\
one & read & two
\end{tikzcd}$$

(b) SBG

Figure 3.2: Sample program and a corresponding SBG

system can provide, the graphs can only be used to detect the presence of bugs on such systems. This is, however, sufficient to drive a generic oracle.

An example of a program and its SBG is shown in Figure 3.2. Note that the function `int read()` is an external one and the SBG only contains the control flow for non-zero return values. The edges resulting from return values equal to zero are missing.

A SBG can be acquired in two different ways. Either a version of the program that is known not to contain faulty components is used or manual verification is employed to ensure the absence of failures during a learning phase. The first approach is only feasible when planning to modify a program (e.g., before a large-scale refactoring), the second does require human effort. Graphs acquired from different runs of the same program can be merged by uniting the edge sets. This allows for incremental learning of an SBG.

The inference of software behavior graphs proved to be straightforward by enabling the SBG oracle to operate in two modes, learning and enforcing. In learning mode, a newly encountered edge was added to the adjacency matrix used to store the graph, in enforcing mode, a failure would be detected. The resulting SBG could then be emitted or the oracle set to enforcing mode. To persist the graph, a code generator was developed that translated the textual representation into a constructor for a pre-filled SBG.

3.3 Driver Interaction Models

The second generic oracle examines the interactions between a system and its periphery. When a driver communicates with an external piece of hardware, this interaction can be observed and used to infer a model describing the "normal" communication between hardware and driver. Note that the model will not necessarily match the complete protocol underlying such communication if the partners only send a subset of valid messages. Interactions can be observed on multiple levels of abstraction and using two different approaches.

The level of abstraction used for observation determines the amount of domain knowledge used to decode the communication. On the lowest level, no information about the communication's contents is used. For example, if interactions using electric signaling are observed, the collected data will be the level changes on the line. On higher levels of abstraction, parts of the communication are decoded. When analyzing communications using a TCP stack, the signals on the wire would be decoded into Ethernet frames, IP datagrams, or TCP packets.

The generic approach to observation uses tools like logic analyzers or network sniffers are used to observe the interactions. These tools might be costly, e.g., a PCI sniffer. Alternatively, instrumentation is introduced into the driver stack that records received and outgoing interactions; the driver is used to decode the signals. However, this approach can not detect all faults inside the driver stack.

The observed interactions might follow complex and non-deterministic logic. This complexity needs to be reduced to permit the inference of models. Finite state machines (FSMs) have been studied extensively as targets for machine learning (e.g., using LearnLib[1] [RSB05]) and can be stored in compact representations. Therefore, interactions were modelled using these. The resulting FSM

$$A = (Q, \Sigma, \delta, q_0, Q)$$

such that

- Q is the set of (inferred) states of the interaction,
- the input set Σ contains the possible communication messages,
- the transition function δ maps $\delta(q, s) \mapsto q'$ if and only if the model assumes s is a valid message in state q and the next state is q',
- the initial state q_0 is the inferred initial state of the interaction, and
- all states are accepting

is called a **driver interaction model (DIM)**.

While these models were developed to localize faults in drivers, the approach can theoretically be generalized to any form of communication. Since a driver has no final state, all states are assumed to be accepting. The states will not necessarily match a "natural" state of the program or the peripheral component.

[1] `https://learnlib.de/`

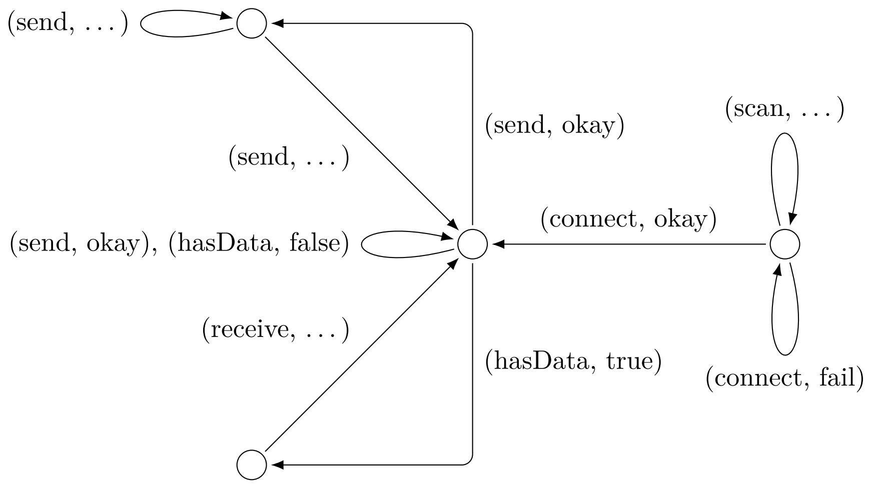


Figure 3.3: A fictional DIM for a wireless chipset driver

Since deterministic finite state machines do not differ in expression strength from non-deterministic ones, DIMs can be represented in both forms. Because deterministic FSMs are easier to handle, all DIMs were determinized for production use.

Figure 3.3 shows the fictional results of high-level instrumentation of a wireless chipset driver. The instrumentation has decoded most of the protocol and has yielded request-response tuples that were used to learn a non-deterministic DIM. Note that "... " signifies "any value" to increase legibility. By examining the DIM, some information about the driver can be inferred, e.g., data is only sent *after* a connection to a network has been established. The learning process apparently never encountered a situation where the network connection was lost; the behavior in that situation is missing from the model.

A DIM can be used as an oracle by applying the transition function for all messages encountered during a run. A trace is considered failing if it contains a message that has no matching transition in the DIM.

Since the DIM can not make a valid transition in this case, normal operation cannot continue and the DIM requires "resetting" to discover more than one failure. A reset threshold i is passed to the oracle. After missing a transition, the oracle remains in the previous DIM state. In this mode, transitions are attempted for new messages, but no failures are reported. When i *successive* transitions succeeded, the oracle returns to normal mode and starts reporting failures again.

Acquisition of a DIM can be performed using the same approaches as described for SBGs. Either runs have to be manually verified as succeeding, or the DIM can only be used for subsequent refinements to a program. However, since the DIM

```
Input: A non-deterministic driver interaction model $A = (Q, \Sigma, \delta, q_0, Q)$
Input: A parameter $k$
Output: A recuded driver interaction model
begin
    repeat
        $m := 0$;
        for $q_s, q_t \in Q, q_s \neq q_t$ do
            if $\exists s \in \Sigma^k : \delta^*(q_s, s)$ is defined , $\delta^*(q_t, s)$ is undefined then
                continue;
            for $q \in Q, x \in \Sigma$ do
                if $q_s \in \delta(q, x)$ then $\delta(q, x) := \{q_t\} \cup (\delta(q, x) \setminus \{q_s\})$;
            for $x \in \Sigma$ do $\delta(q_t, x) := \delta(q_t, x) \cup \delta(q_s, x)$;
            $Q := Q \setminus \{q_s\}$;
            $m := 1$;
    until $m = 0$;
    return $A$;
```

Listing 3.1: Modified k-tails algorithm

does not examine the program's internal structure, it can remain usable even after major rewrites to the program.

3.4 Learning DIMs

Inferring a driver interaction model is a CPU- and memory-intensive task that requires multiple runs of the target program. Performing this inference was not feasible using an embedded system, therefore, DIM learning was performed on a desktop PC.

To infer a DIM from a set of DIM event sequences, a FSM has to be constructed using passive positive-example learning. Active learning is impossible for the task at hand since the system can not be queried for valid sequences – a valid sequence is one generated by the fault-free system. Negative examples can not exist for the same reason.

This excludes most advanced FSM inference algorithms. In a review of existing algorithms, Murphy suggests the k**-tails** algorithm [BF72] for this scenario [Mur95]. k-tails attempts to iteratively merge equivalent states of the automaton. States are considered equivalent if their k**-tail**, i.e., the set of words of length k that are

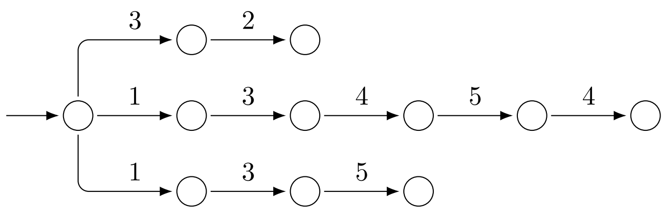


Figure 3.4: Initial FSM for three DIM event sequences

accepted starting from the state, are equal. When merging, all transitions to the source state are moved to the target state, the source state is removed.

The equivalence condition was relaxed to a subset relation. As long as a source state's k-tail was *contained* in a target state's k-tail, a merge was performed. The resulting algorithm is depicted in Listing 3.1. Some edge cases are omitted: the initial state can not be a merge source and states with no k-tail were exempted from merging as both sources and targets.

To create an initial valid FSM, every DIM event sequence is transformed into a chain of automaton states using the events as edge labels, with the initial state being identical among all chains. The resulting models are valid, although not yet useful as an oracle. The result for the event chains

$$\begin{gathered}\langle 1,3,5\rangle \\ \langle 1,3,4,5,4\rangle \\ \langle 3,2\rangle\end{gathered}$$

is illustrated in Figure 3.4.

The performance of this algorithm proved to be unacceptable. Real-world DIM event sequences resulted in automatons containing millions of states to be created. Pairwise checking subsequently caused merge rates to drop below one merge per second.

However, the initial automata have several properties that permit an improved version of the algorithm to be created. First, every state, except for the initial state, has exactly one successor that is not the initial state. Therefore, the state must have a single k-tail. (The final k states have none and are ignored.) When merging, the target state's k-tail can not grow due to the equivalence criterion.

Since the initial automaton is constructed from event chains, the k-tail for a new state can efficiently be read from the next k elements in the input sequence. This allows to efficiently decide when a new state would be merged into a previous one and create an edge to the merge target instead. The k-tail of the initial state is constructed differently and contains the first k elements of *all* input sequences.

```
**Input:** A set of sequences of DIM events $\{S^1, S^2, \dots, S^n\}$ from an alphabet of
        messages $\Sigma$
**Input:** A parameter $k$
**Output:** A recuded driver interaction model
1  begin
2      $q_0 \coloneqq$ a new state;
3      $Q \coloneqq \{q_0\}$;
4      $K : \Sigma^k \to Q \cup \{\varepsilon\}$ and $K(x) \coloneqq \varepsilon$ for all $x$;
5      for $i \in \{1, \dots, n\}$ do $K(\langle S^i_1, \dots, S^i_k \rangle) \coloneqq q_0$;
6      for $i \in \{1, \dots, n\}$ do
7          $q \coloneqq q_0$ for $j \in \{1, \dots, \mathrm{len}(S^i) - k\}$ do
8              $t \coloneqq \langle S^i_{j+1}, \dots, S^i_{j+1+k} \rangle$;
9              $q' \coloneqq K(t)$;
10             if $q' = \varepsilon$ then
11                 $Q \coloneqq \{q_{i,j}\} \cup Q$;
12                 $K(t) \coloneqq q_{i,j}$;
13                 $q' \coloneqq q_{i,j}$;
14             $\delta(q, S^i_j) = \{q'\} \cup \delta(q, S^i_j)$;
15             $q \coloneqq q'$;
16         for $j \in \{\mathrm{len}(S^i) - k + 1, \dots, \mathrm{len}(S^i)\}$ do
17             $Q \coloneqq \{q_{i,j}\} \cup Q$;
18             $\delta(q, S^i_j) = \{q_{i,j}\} \cup \delta(q, S^i_j)$;
19             $q \coloneqq q_{i,j}$;
20     return $(Q, \Sigma, \delta, q_0, Q)$
```

Listing 3.2: Optimized k-tails algorithm

The optimized algorithm in Listing 3.2 exploits these facts to create a k-tails-reduced automaton directly from input sequences. The algorithm initially computes the k-tail for the initial state since it can not be trivially derived (Line 5). Then, the input sequences are read.

For every event, the k-tail of a *potential next* state is determined (Lines 9 and 10). If a state with this k-tail already exists, a transition to it is added and it is set as the current state (Lines 14 and 15). If not, a new state is created and stored as canonical for the k-tail, then, a transition is added and the current state updated (Lines 11 to 15).

Finally, the states with no k-tail are added as a sequence after the last regular state (Lines 16 to 19). The resulting structure is subsumed into the remaining FSM after determinizing the non-deterministic automaton.

3.5 Timer-Based Transactions

A generic approach to determine transactions using a timer was presented by Abreu, Zoeteweij, and Gemund. Every second, the current spectrum was exchanged for a new one and scheduled for integration [AZG06]. This implicitly results in transactions being 1-second-fragments of the trace.

For this thesis, the fragments were measured using a configurable interval and a timer running on the system responsible for data analysis. The resulting transaction detector is completely agnostic of the program being examined. As a minor downside, if thread separation is active, transactions are not detected for a specific thread, but for all active ones.

While more sophisticated approaches to transaction detection are certainly possible, they were not investigated due to the successes achieved with this technique in the literature.

3.6 Failure-Specific Analysis

Oracles can mitigate the inability of SBFL to reliably detect more than a single fault in a single iteration. This is not an issue in classic SBFL: one fault will be repaired, analysis will be repeated and the next fault will be located.

In long-running systems, repeated analysis-repair-cycles are difficult to implement and require larger amounts of manual work than necessary. Therefore, it would be preferable if good SBFL results for multiple failurescould be obtained in the hope that they uncover multiple faults.

Failures detected using multiple oracles can easily be distinguished by the detecting oracle. Additionally, the oracle can also provide extended information about the failure. For example, an oracle detecting high reaction times might obtain the identity of the slow component. This is called **failure indexing**; again, the feature was made configurable and both settings were compared.

When distinguishing failures, a separate SBFL report per detected failure is prepared. If several failures point to the *same* fault, this approach might dilute the gathered data instead of improving the analysis process.

Chapter 4

AOP-Based Instrumentation

The previous chapters have left open the question of data acquisition. The examined program needs to be instrumented to report on the executed components in order to perform SBFL and acquire the additional data required by SBG and DIM oracles.

For this thesis, instrumentation was performed using aspect-oriented programming and the AspectC++ framework. In this chapter, both the concept of aspect-oriented programming and AspectC++ itself are introduced in detail.

4.1 Basic Concepts

In many programming disciplines, parts of a system's function can not cleanly be encapsulated into a component. These parts are called **aspects** [Kic+97]. **Aspect-oriented programming (AOP)** [Kic+97] extends programming disciplines to allow the seperation of aspects into separate source code components. Common aspects that are extracted into into a dedicated component include method-level access control, logging, database transaction management and synchronization.

When extracting an aspect, **join points** [Kic+97] are those parts of the program the aspect needs to affect. For example, when extracting the aspect of trace logging, i.e., writing the name of the called function to a log file after each function call, the join points would be all function calls in the program.

Since manually specifying each and every join point would add a severe maintenance burden – the list would have to be kept synchronized with the actual program – sets of join points are specified instead. These sets are called **pointcuts** [Kic+01] and are usually described by the properties of the contained join points, e.g., names of called functions, containing classes etc.

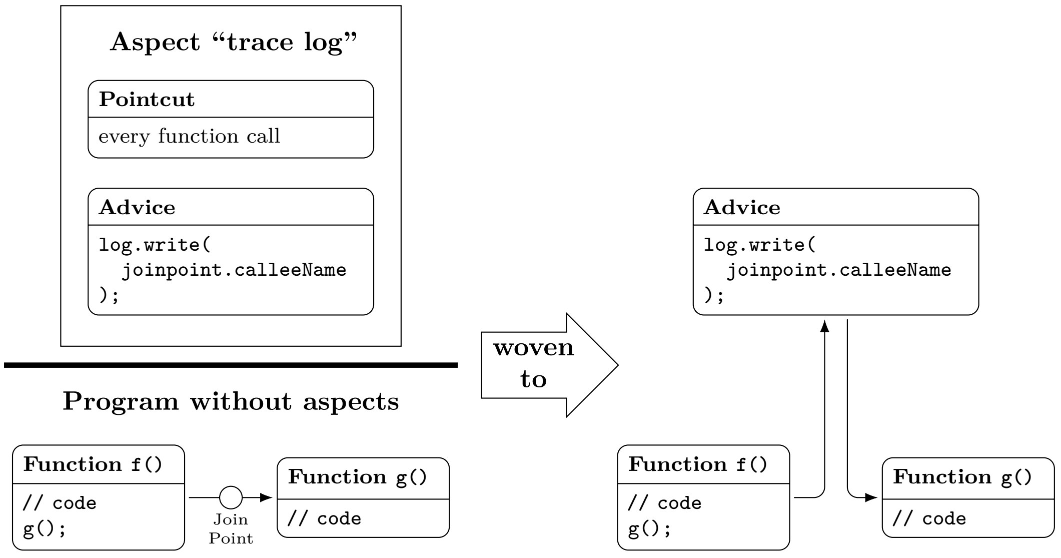


Figure 4.1: Aspect weaving introduces an advice into the control flow

The logic that needs to be executed at one or more pointcuts is called an **advice** [Kic+01]. An advice is often written in a dialect or extension of language the target program is developed in.

To continue the example, the trace logging aspect would store a handle for the log file. It would execute at a pointcut that contains all function calls; on each execution, the name of the called function would be written to the log file.

Support for AOP is provided by an **aspect weaver** [Kic+97]. This component is responsible for modifying a program so that at defined pointcuts, an advice is introduced into the execution. The interaction of the various components is illustrated in Figure 4.1: the aspect weaver modifies the program's control flow to include the advice code. Aspect-oriented programming encompasses more techniques than those introduced here; since these bear no relevance to this thesis, they are omitted.

4.2 AspectC++

For the analysis framework, the AOP framework **AspectC++** [SGS02; FBS17] was used. AspectC++ uses standard C++ for the aspect-less program and an extended C++ dialect – the AspectC++ language – to define aspects and pointcuts. The framework also provides an aspect weaver for the language, the AspectC++ compiler. Furthermore, the framework implements several advanced features not required for this thesis.

```
aspect TraceLogging {
    pointcut everyCall() = "%";
    static logging::Logger log;
    advice execution(everyCall()) : before() {
        log << "entering "
            << tjp->signature()
            << "\n";
    }
};
```

Listing 4.1: Trace logging aspect implemented in AspectC++

The logic of advices is implemented in standard C++; to access information about the current join point, the object `tjp` and the class `JoinPoint` are injected into the code. Pointcuts are specified using

- either a matching language that can match functions by name, parameter types, return type, and other properties,
- by defining custom C++ annotations that can be applied to the program's code, or
- a combination of both approaches.

The weaving step replaces all language extensions and outputs standard-compliant C++ code that can then be compiled using a standard C++ compiler. Listing 4.1 shows a trace logging aspect written in AspectC++ that outputs the function's signature after each call of a function or method in the program.

During weaving, the AspectC++ compiler collects metadata about the project in an auxiliary XML file. This information includes the logical structure of the program (namespaces, classes, methods etc.) and information about the weaving performed by the compiler. Each join point is assigned a unique, numerical, sequential identifier, the **join point identifier (JPID)**, that can also be programatically accessed by an aspect. Using this information, a mapping from JPIDs to function and method signatures can be established. This thesis uses the mapping to store signatures compactly.

Chapter 5

Implementation

This chapter describes the architecture and high-level implementation of the analysis framework developed for this thesis. Also, the communication protocol for tracing is described and the data structure used for storage explained.

The components that needed to be implemented for deployment on a specific platform will be outlined in the next chapter.

5.1 Architecture

The analysis framework is set up on two separate devices. The first machine hosts analysis software and will attempt to recognize failures and localize the faults responsible for them; this is called the **monitor**. The second machine hosts the program that is suspected of harboring faults; this is called the **target**. An sketch of their interaction is depicted in Figure 5.1.

Retrofitting of tracing capability into the target program was performed using the AspectC++ compiler to weave in a tracing aspect. This approach requires no changes to the program code, only to the build system. While several implementations of SBFL have instrumented code at the statement or block level, this is not possible with AspectC++. Therefore, the trace data only contains information about function and method calls.

The acquired trace data is immediately transmitted to the separate, physically isolated monitor to protect against data corruption and crashes. Since the tracing aspect can not be guaranteed exclusive access to the transmission hardware on all combinations of hardware and operating system, some synchronization with the target OS is required for this step.

This monitor uses oracles and transaction detectors to segregate the trace into transactions and identify failures. This information is stored in aggregate matrices. On demand, this data will be condensed into an SBFL report using one or more

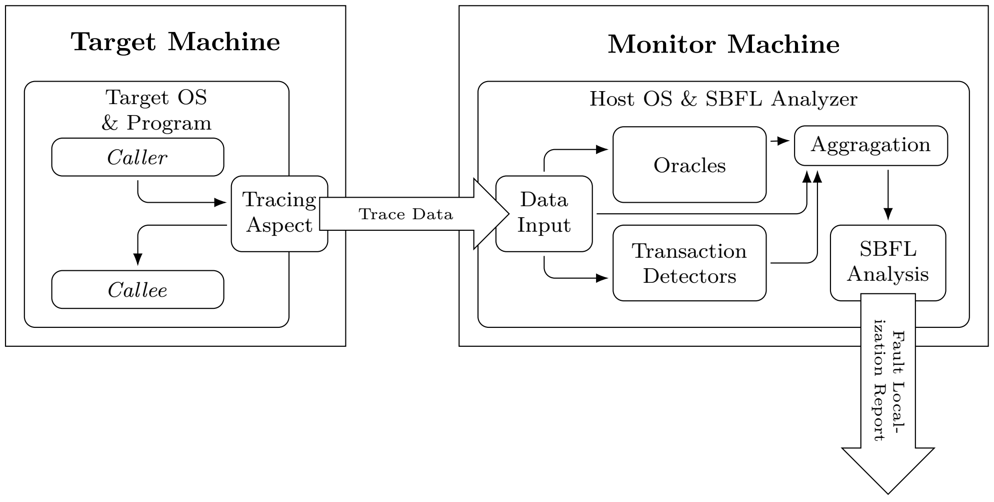


Figure 5.1: High-level architecture of the analysis framework

suspiciousness metrics. This report is output to a human operator, who can use it to ascertain the presence of failures and, if necessary, localize the underlying faults in the target program.

5.2 Framework Architecture

The components of the developed framework can be separated into three layers. Some are independent of the target platform, some provide the necessary "glue" between those and the OS and some are alterations that had to be made to the target OS itself. An overview of the components is shown in Figure 5.2; detailed explanations follow.

5.2.1 Tracer

The tracer is the aspect responsible for gathering information about function and method calls in the target program and is woven into most calls. Some components that are called during early boot, responsible for context switches or that are called too frequently to feasibly trace them have to be excluded to maintain system stability.

This makes the tracer slightly platform-aware, though not -dependent. During deployment, the components that could safely be traced were quickly identified using trial and error; for programs running in a managed environment, it is assumed that all components can safely be traced. To allow the target program to suspend tracing, a global variable was introduced.

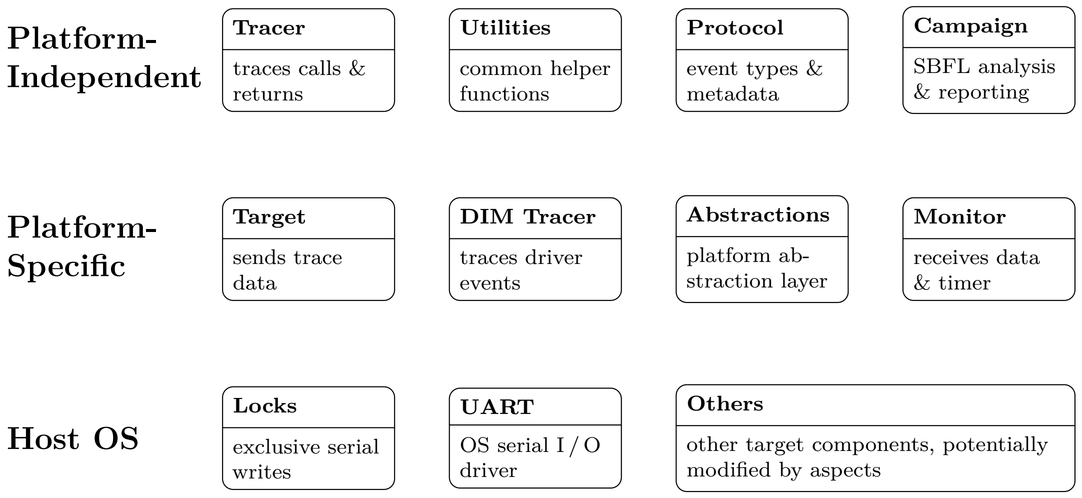


Figure 5.2: Component structure of the analysis framework

Tracing itself is composed of three parts: first, the called function or method is identified by the current join point. Second, the parameters are classified using the interface described in Section 5.2.3. Third, the current thread of execution is identified to enable thread separation (see Section 2.6). For operating systems that do not use threads in the classical sense, the id of the active CPU can be substituted. Finally, the gathered information is sent to the monitor.

A simplified version of the tracing aspect is shown in Listing 5.1. The actual implementation requires additional template metaprogramming to extract the parameters, which has been omitted in the interest of brevity. In the example, only the first parameter is extracted. Also, the whitelist of safely traceable components has been reduced to a single entry.

The functionality to communicate with the monitor, obtain the thread identifier and suppress tracing are platform-dependent. Therefore, they are not implemented in the aspect, but are provided by the target component described in Section 5.2.5.

To create and verify an SBG, a trace is not sufficient, since information about function returns is also required to correctly identify callers. The code for tracing returns is nearly identical to that for tracing calls and is also located in the tracing aspect.

5.2.2 Utilities

This component collects non-SBFL-related functions and classes that are required for the other components. This includes features that are provided by a C++ standard library since embedded systems do not necessarily offer one.

```
aspect SBFLTracer {
    pointcut whitelist() = "% %Debug::...::%(...)";
    advice execution(whitelist()) : before() {
        if (!tracingEnabled()) { return; }

        uint8_t classifiedParameter;
        if (canClassify<typename JoinPoint::Arg<0>::ReferredType>
                    ::value) {
            classifiedParameter =
                classify(*(tjp.template arg<0>()));
        } else {
            classifiedParameter = 0;
        }

        SBFLTraceEvent event(TraceWriter::threadID(),
                             JoinPoint::JPID,
                             classifiedParameter);
        TraceWriter::record(event);
    }
};
```

Listing 5.1: Simplified version of the tracing aspect

5.2.3 Protocol

The protocol component provides an object-oriented interface for reading and creating messages using the communication protocol described in Section 5.3. It also contains support code for the parameter classification described in Section 2.5 and stores the mapping from JPIDs to signatures described in Section 4.2.

A generic way to classify parameters was realized using template metaprogramming as demonstrated in Listing 5.2. For a (statically determined) type `T`, the expression

`canClassify<T>::value`

is resolved to a boolean at compile time, while

`classify<T>(t)`

will classify the value `t` of type `T` at runtime.

5.2.4 Campaign

The campaign component is completely platform-independent and implements the SBFL analysis algorithm. It receives trace information and driver interaction

```
template<typename T>
struct canClassify { static const bool value = false; };
template<typename T>
uint8_t classify(T const &value) { return 0; }

template<>
struct canClassify<int> { static const bool value = true; };
template<>
struct canClassify<const int> { static const bool value = true; };
template<>
uint8_t classify (int const &value) {
    return (value > 0) ? 1 : (value < 0) ? 2 : 0;
}
```

Listing 5.2: Classification support for the `int` type

events and uses this information to detect failures and compute likely locations of the underlying faults. Additionally, the campaign can receive special commands, e.g., to show a fault localization report to a user. The internal layout of the component is depicted in Figure 5.3.

The central sub-component of the campaign is the event receiver. This class is notified about incoming trace events, DIM events and commands and is responsible for handling them.

If a trace event is received, the information is first passed to the oracles and transaction detectors for analysis. Since all implemented oracles and transaction detectors do not use information about future events, they can immediately decide on the presence of a failure or the beginning or end of a transaction.

The called component and the failure information are then passed to the spectrum store, which then integrates the information into a spectrum. If thread separation is enabled, one store per thread is maintained and the trace information is sent to the corresponding store. Otherwise, only a single store is created. If MCSHS are used, stores are responsible for maintaining and storing the method call sequences. If the end of a transaction is detected, the spectrum store is again notified. It then passes its current spectrum and all failure data to the aggregate store and resets itself to empty state.

The aggregate store, having received a spectrum and failure data, updates the aggregate matrices for all components in the spectrum. To reduce memory requirements, only the second row of the aggregate matrices, i.e., the number of succeeding and failing runs the component participated in, is stored. Together with a global count of failed and successful transactions, the complete matrix can be recomputed.

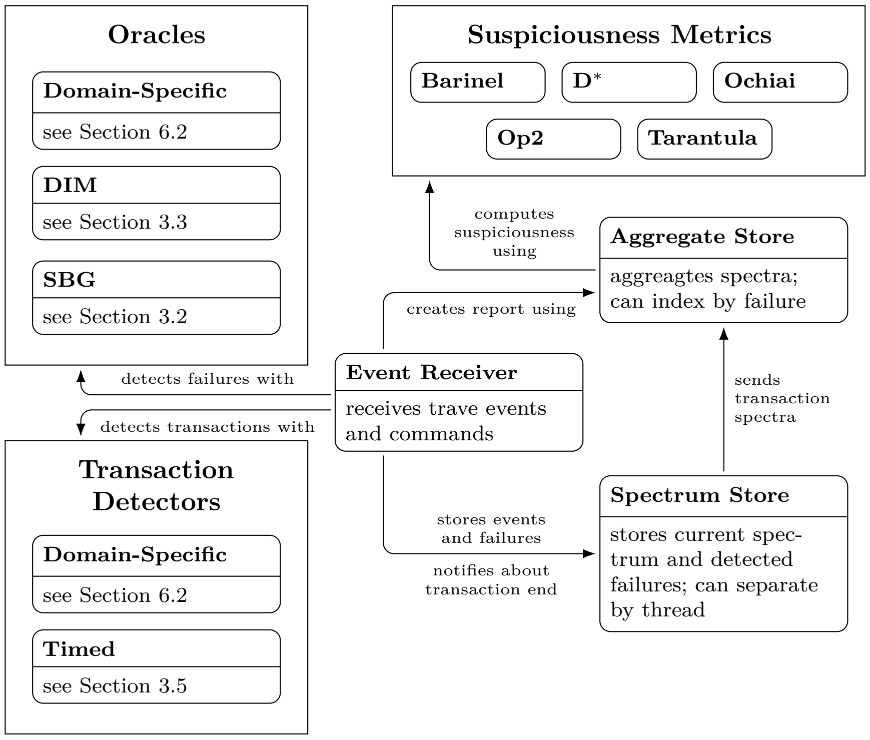

Oracles
Domain-Specific
see Section 6.2
DIM
see Section 3.3
SBG
see Section 3.2
Suspiciousness Metrics
Barinel
D*
Ochiai
Op2
Tarantula
computes suspiciousness using
Aggregate Store
aggreagtes spectra; can index by failure
creates report using
detects failures with
Event Receiver
receives trave events and commands
detects transactions with
sends transaction spectra
Transaction Detectors
Domain-Specific
see Section 6.2
Timed
see Section 3.5
stores events and failures
notifies about transaction end
Spectrum Store
stores current spectrum and detected failures; can separate by thread

Figure 5.3: Structure of the campaign component

If failure indexing is enabled, the store allocates the per-failure aggregate objects lazily. Only if a failure is encountered, a new object is created and information about previous transactions filled in. This is made possible by storing a utility aggregate that records every spectrum as succeeding. This data is copied into the new object after creation. If failure indexing is disabled, only one aggregate is required and lazy allocation is not necessary.

When the event receiver is notified about a DIM event, the DIM oracle is queried about a potential failure. If it detects one, and thread separation is enabled, all thread spectra are notified since the event can not be associated with a specific thread of execution.

The creation of a SBFL report is handled by the aggregate store, which uses a library of suspiciousness metrics and provide scores using each of the supplies metrics. The design is modular to allow quick addition of new metrics for examination.

5.2.5 Target

The target component is a platform-dependent component responsible for sending events to the monitor. This is done using the operating system's UART driver.

```
aspect GPIODIMTracer {
    advice execution(gpioWrite()) : before() {
        if (!sbfl::target::tracingEnabled()) { return; }

        uint32_t bank = gpioDRToBank(*(tjp->arg<0>()));
        uint8_t pin = *(tjp->arg<1>());
        uint8_t value = *(tjp->arg<2>()) ? 1 : 0;

        DIMTraceEvent event(64 * (bank - 1) + 2 * pin + value);
        TraceWriter::record(event);
    }

    advice execution(gpioInterrupt()) : before() {
        if (!sbfl::target::tracingEnabled()) { return; }

        DIMTraceEvent event(64 * interruptBank);
        TraceWriter::record(event);
    }
};
```

Listing 5.3: Simplified version of a DIM tracing aspect for GPIO

To avoid an infinite loop in the instrumentation logic, the driver can not be instrumented.

Since sending data requires uninterrupted access to the UART driver, a priority mechanism had to be introduced that is described in Section 5.2.9. The component also provides an interface to temporarily suppress tracing and is used to obtain the identity of the current thread.

5.2.6 DIM Tracer

Similar to the primary tracer, the DIM tracer records driver interaction events. Since this tracer gathers its data by instrumenting the low-level GPIO driver inside the target OS, it can not be implemented in a platform-independent way. Fortunately, finding the correct methods for instrumentation proved to be both fast and simple.

Tracing driver interactions requires far less information than gathering data for SBFL itself, since neither parameters nor threads need to be handled. Instead, an ID is assigned to each possible event; the ID is then transmitted to the monitor.

A simplified version of the tracer is shown in Listing 5.3. This aspect omits the definition of the pointcuts, constants and some utility functions that are of minor interest.

5.2.7 Abstractions

Since a standard library is not necessarily present on an embedded system, standard C++ types such as `uint32_t` might only be provided in a proprietary header file. This same issue arises for text output streams and the memory allocation mechanism.

To avoid the introduction of platform-specific code in higher layers, the abstractions component provides an adapter for these dependencies. It provides

- a header that includes the system-specific type header,
- a header that provided macros wrapping the host operating system's memory allocator and
- an output stream for text behaving similar to the `<iostreams>` library.

This component requires rewriting when porting the system to another platform.

5.2.8 Monitor

The monitor component is responsible for receiving timer and UART input events from the operating system's event dispatcher and forwarding them to the campaign, while handling protocol decoding using the respective component.

For experimentation purposes, the component was given a second responsibility. When using timed transaction detection, experiments become impossible to reproduce exactly due to the timer introducing non-deterministic transaction boundaries. To reintroduce determinism, the OS timer can be substituted with commands sent by an external timer. The synthetic timer event is indistinguishable from a locally-sourced one, allowing for deterministic replay of trace data when combined with timer information.

Two other commands handled by the monitor component disable and enable UART output from the monitor system If the monitor's OS writes debug output to the serial interface, disabling the UART avoids accidentally sending data to the target system. This required patches to the drivers themselves, although a reimplementation as an aspect might avoid the need to modify the OS.

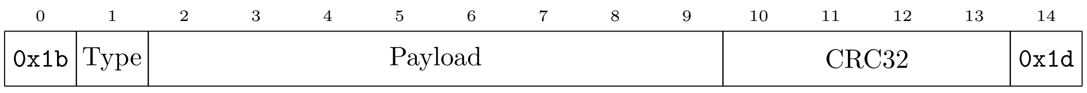


Figure 5.4: Structure of an event package

5.2.9 Locks

The locking mechanism requires modifications to the operating system itself. All components that access the UART driver, including the framework's target component, must acquire a lock. The OS needs to modified to introduce this behavior.

This avoids interference between an event and other output and allows shared usage of the serial interface while guaranteeing the integrity of sent events. Section 5.3 details how the communication protocol provides for separation of the events and other output.

5.3 Communication Protocol

To communicate trace events, DIM events and commands to the monitor, serial communication over a UART was used. The target, however, may also use the serial interface to communicate with the user, receive commands and emit diagnostic messages. The protocol therefore has to co-exist with regular input and output. All data sent by the target itself is assumed to be text data, i.e., a sequence of characters from the ASCII character set [Cer69].

Since no send buffer was maintained on the target device to avoid corruption of the buffered data, events had to be sent synchronously. This causes a slowdown of the system proportional to the UART speed and the size of a message. Messages therefore had to be as small as possible.

To delimit an event package, two ASCII control characters, escape (`0x1b`) and group separator (`0x1d`) were chosen since they are not commonly used in ASCII-encoded documents.

Since, contrary to the previous assumption, the target might emit these characters or the payload might be corrupted during transmission, an additional checking mechanism was introduced. A CRC32 checksum [PB61; HBL75] was chosen due to guaranteed detection of single-byte errors and a high resilience against transmission errors in general. The checksum is only computed over the type and the payload fields since an invalid start or stop byte causes the package to be ignored anyway.

This mechanism allows to separate text and trace output cleanly when reading from the UART. The monitor component itself discards both invalid event packages and other data. During experimentation, no invalid packages were observed.

5.4 Data Structures

Since the developed framework must operate in an embedded environment, it needs to limit its memory usage. For most components, the memory requirements were satisfied using statically-allocated memory and small data structures allocated on the stack. Since the amount of memory required for these was insignificant, they were not examined further.

However, the campaign component needs to store MCSHS and index them by failure. The naive approach to storing a spectrum over n components would be an array of size n; aggregate matrices could be stored in a similar array. For less than five hundred functions in a small program, this approach is feasible.

If parameter classification is enabled, every function corresponds to a set of variants. For five parameters (the greatest number encountered during implementation) and a maximum of seven classifiers, this results in a maximum of $7^5 = 16\,807$ variants. This would result in 8 403 500 array entries, which can still be stored in a few megabytes. However, there are $8\,403\,500^2 = 70\,618\,812\,250\,000$ method call sequences of length two, requiring more than 70 terabytes of storage for a single spectrum. Therefore, a more compact approach to storage is necessary.

Before discussing this approach, the nature of detected failures can be examined:

- The DIM oracle triggers due to a missing transition; since the layout of the DIM does not necessarily match real world concepts, no failures can be distinguished.
- The SBG oracle triggers due to a pair of calls that describe the missing edge. The set of potential failures is the set of call sequences of length 2.
- Domain specific oracles[1] are triggered by single, specific calls that also describe the failure. The set of potential failures therefore is the set of calls.

Next, the data that is stored by a campaign can be studied. Given a method call sequence length of k, the following objects require a storage scheme:

Spectra are a mapping from call sequences of length k to booleans.

Aggregate matrix storage can be implemented as a mapping from call sequences of length k to pairs of counters.

[1]An example will be given in Section 6.2.

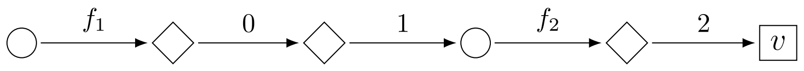


Figure 5.5: Trie storage for a single entry

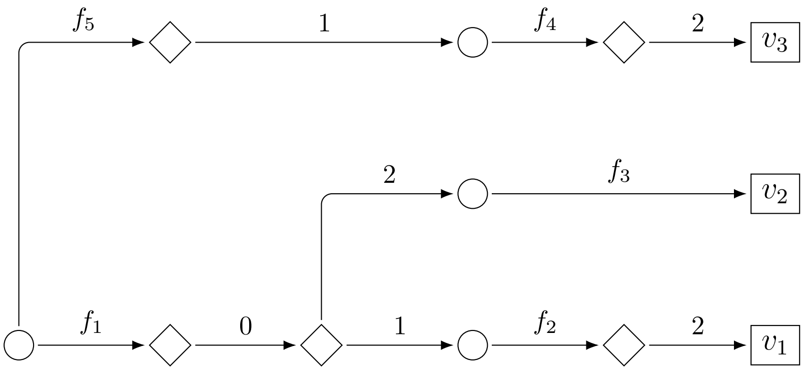


Figure 5.6: Trie storage for multiple entries

The failure indices are a mapping from call sequences of length 0, 1 and 2 to aggregate matrix storages.

It is apparent that the only data structure required is a key-value-mapping from call sequences to arbitrary objects. The call sequences are not fully random, but stem from the trace produced by the target. This means there is only a limited set of successors that can follow a call. To store these mappings efficiently, a modified version of tries [Bri59; Fre60] was used.

First, the storage of a single value will be explained. Assume that a value v is stored for a sequence of two calls. Each call contains a function and 0 to n parameter representatives, depending on the function's arity. As an example, let the sequence be $\langle f_1(0,1), f_2(2)\rangle$. This sequence would be stored as a graph, using a chain of nodes. Every edge corresponds to a function or its parameters, as illustrated in Figure 5.5.

When storing multiple values, common key prefixes are merged in the trie. A trie storing the mappings

$$\begin{aligned}
\langle f_1(0,1),\ & f_2(2)\rangle \mapsto v_1 \\
\langle f_1(0,2),\ & f_3()\rangle \mapsto v_2 \\
\langle f_5(1),\ & f_4(2)\rangle \mapsto v_3
\end{aligned}$$

is depicted in Figure 5.6. To implement this structure for C++, three types of nodes are used:

Value nodes store a value object. They are the leaves of the trie. In Figures 5.5 and 5.6, they are represented by rectangles.

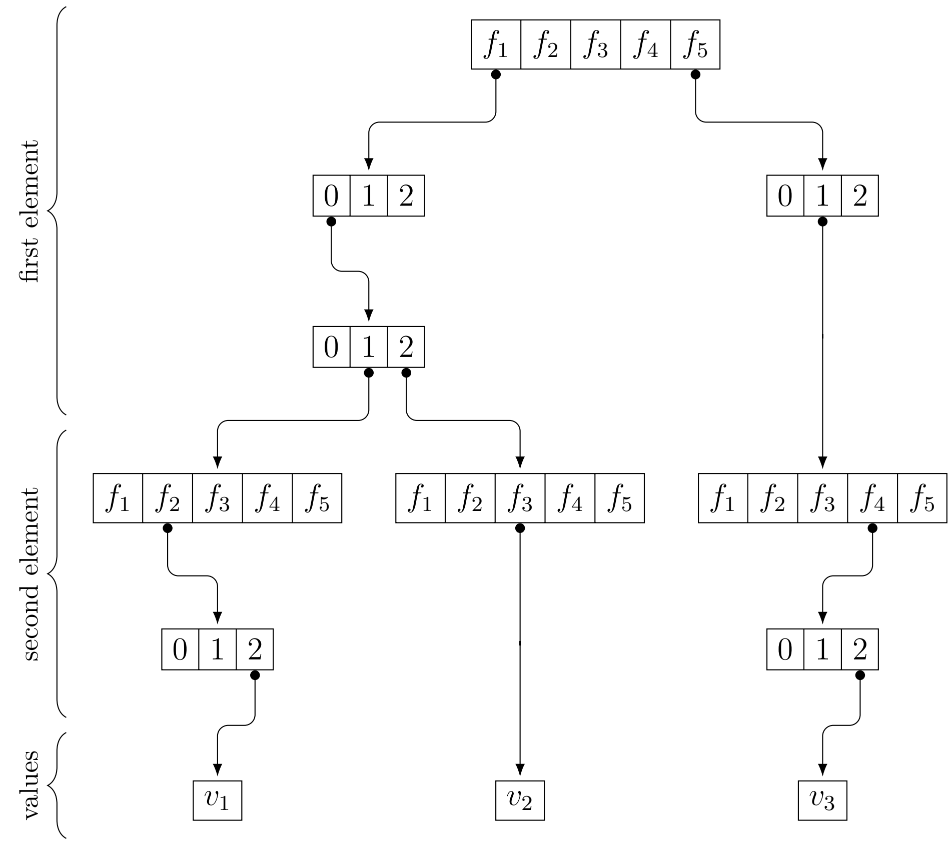


Figure 5.7: In-memory structure of trie storage

Function nodes store an array of pointers to nodes. The array's size is the number of instrumented functions in the target program. Since AspectC++ assigns sequential JPIDs, they can be used as function identifiers and as indices when accessing the array.

Each array element contains a `nullptr` if no value is stored below it or a pointer to another node if there is. If the function with the respective JPID has parameters, that pointer will reference a parameter node, if not, the next function or value node. In Figures 5.5 and 5.6, they are represented by circles.

Parameter nodes also store an array of pointers to nodes; the array has size equal to the largest number of classifiers for any type. The representatives are used as indices.

Each element again contains a `nullptr` if no value is stored below it or a pointer to another node if there is. If the parent function has another parameter, it will reference a parameter node, if not, the next function or value node. In Figures 5.5 and 5.6, they are represented by diamonds.

The resulting in-memory structures are shown in Figure 5.7 for five functions and a maximum representative of 2. The resulting nodes form one “layer” per element in the method call sequence serving as a key. This results in a fixed number of layers. Below the final layer, the values are stored.

The maximum storage required for n_v values, n_f functions with at most n_p parameters, n_r representatives and sequences of length k is

$$n \cdot r \cdot (n_f + n_r \cdot n_p).$$

This case only occurs if all method call sequences used as keys diverge in the first element, i.e., if every key used for storage diverges in the first function. This scenario is highly unlikely for the intended usage of the tries and a sufficient number of stored mappings. The storage requirements for real-world input will be examined empirically in Section 7.5.

Looking up a value in the storage requires traversal of the trie. This can be performed in linear time with respect to the (constant) depth. Storing new values is not possible using static memory, instead, the operating system’s implementation of C++ dynamic memory API (`new` and `delete`) was used to allocate new nodes, making support for dynamic memory hard requirement.

Chapter 6

Deployment on CyPhOS

While the bulk of the analysis framework operates independently of the host platform, some adapter components need to be implement to port the framework to a platform. For this thesis, the framework was deployed on the operating system CyPhOS.

This chapter introduces CyPhOS and some of its components, describes the platform-specific parts developed for the analysis framework and outlines some of the issues encountered during implementation.

6.1 Host Platform

This section gives an overview of the software and hardware that were used as a deployment platform for the analysis framework. These include CyPhOS itself, its motor control component, EMSComponent, and the EMSBench software used to simulate a motor.

6.1.1 CyPhOS

The operating system **CyPhOS (real-time operating system for cyber-physical systems)** [BS15] is designed to provide predictable timing behavior during execution. This is realized by separating the system into several components, **operating system components (OSCs)**, that interact over an event bus. By loading OSCs into a partition of the memory cache before execution, it is guaranteed that no accesses to main memory must be performed. This avoids the randomness in timing behavior encountered when interacting with normal RAM. By eliminating non-cached RAM accesses, the OS can provide real-time characteristics even on non-purpose-built hardware and while running applications in parallel on multiple cores.

Figure 6.1: Manufacturer's photo of a Wandboard Quad[1]

CyPhOS is written in C++ and already incorporates the AspectC++ compiler into its build process. While the real-time characteristics were not exploited for this thesis, it provided a convenient platform for both the gathering of data and its analysis.

CyPhOS itself was deployed to a Wandboard Quad (depicted in Figure 6.1), a development board hosting an i.MX6 CPU providing the ARM Cortex-A9 instruction set and 2 gigabytes of RAM.

6.1.2 EMSComponent

To validate the real-time capabilities of CyPhOS, Schulte-Althoff implemented an **engine management system (EMS)** running on CyPhOS. EMSs control ignition and fuel injection for a motor's cylinders and are a stock component of modern cars.

The component, called EMSComponent [Sch17], was designed to interact with a (simulated) motor under extreme system loads. Its responsibilities include interrupt handling and timing measurements that needed to be completed under time pressure. Even under a high system load and for 20 000 simulated revolutions per minute, the component performed with high accuracy. The original setup is depicted in Figure 6.2. It uses a connection board between the trace generator

[1]Retrieved from `https://www.wandboard.org/products/wandboard/WB-IMX6Q-BW/`.

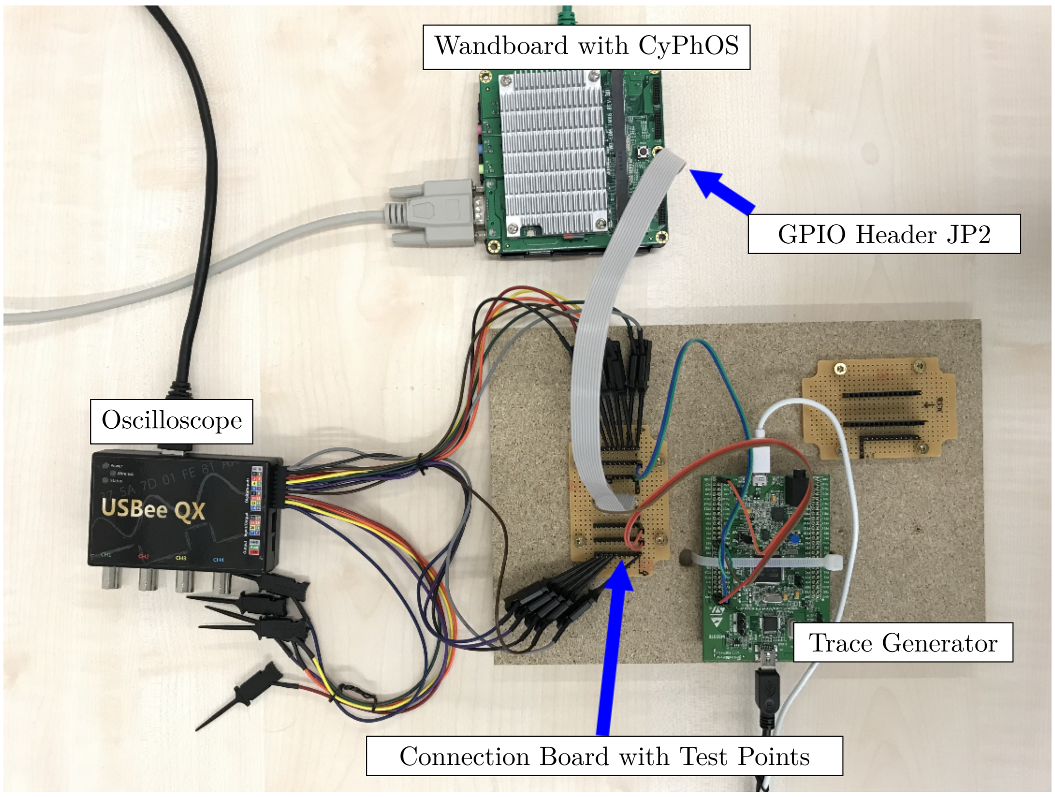


Figure 6.2: The experimental setup for the EMSComponent (translated, illustration from [Sch17])

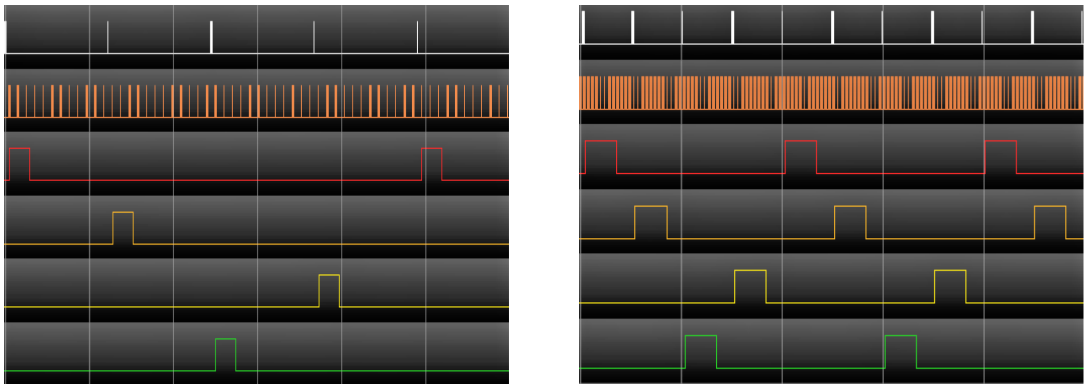
and the Wandboard; an oscilloscope can be connected to the board to monitor data traffic. This setup was reused for this thesis.

The EMSComponent was chosen as a test application for evaluation since it does not interfere with the communication medium chosen for trace data (i.e., UART) and implements algorithms that are sufficiently complex to be sensitive to a wide selection of faults.

6.1.3 EMSBench

To simulate a motor for the EMSComponent to control, EMSBench[2] [KU15] was used. This software can either act as an EMS or simulate a motor to aid in the development of EMSs. The second mode is called trace generation. In this mode, EMSBench does not "receive" control signals; the trace will be generated independent of any input.

[2]`https://www.informatik.uni-augsburg.de/en/chairs/sik/research/running/emsbench/`



(a) injection signals (b) ignition signals

Figure 6.3: Interrupts generated by the trace generator (lines 1 and 2) and injection and ignition signals sent by the Wandboard (lines 3–6) (illustrations from [Sch17])

EMSBench was used by Schulte-Althoff to develop and evaluate the EMSComponent. The software was deployed on a STM32F4-Discovery development board, a low-power development board featuring an ARM Cortex-M3 CPU. Figure 6.3 shows recordings of the expected communication between the EMSComponent and EMSBench, collected under no system load. Note that the signals are repeating due to the motor's operational cycle.

6.2 CyPhOS-Specific Oracles and Transaction Detectors

To provide a baseline for evaluation and test the oracle and transaction detector system, a domain-specific oracle and a domain-specific transaction detector were developed for CyPhOS and the EMSComponent. These required detailed knowledge about CyPhOS to implement, indicating a high up-front workload.

If a fault in CyPhOS causes a CPU exception handler to be invoked, some diagnostic information is emitted over the serial interface; then, the CPU is reset. To emit diagnostic information, CyPhOS' CPU exception handler invokes the function

```
void printCrashContext(dword_t *sp).
```

This call is intercepted by the trace aspect, which transmits the information to the monitor. This is sufficient for an oracle to detect a failure.

Detection of transactions was realized by exploiting CyPhOS' component system. A context switch to an OSC is performed by the method

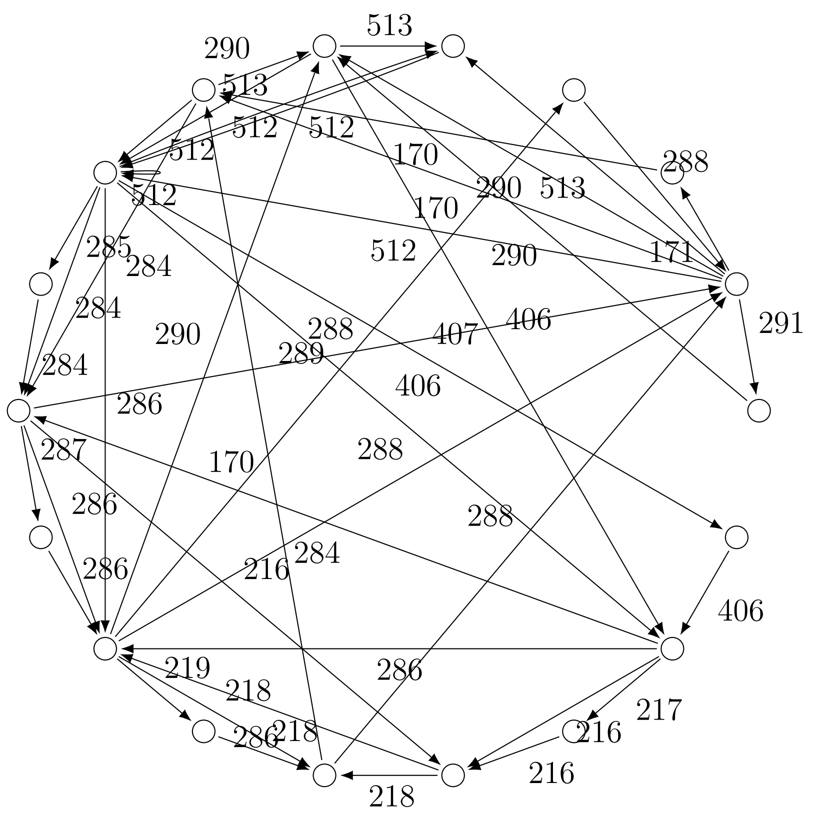


Figure 6.4: DIM learned with $k = 1$

```
void EventHandler::activateOSC(EventTask *event).
```

Similar to the oracle, calls to this method signal the beginning of a new transaction. A limitation of this approach is that code outside of OSCs is not separated into transactions correctly. This affects some system code and all interrupt handlers.

6.3 Generic Oracles and Transaction Detectors

The canonical SBG used for the evaluation was acquired in a single run by deploying the target program, recording the wandboard's output using a PC, deploying the monitor and replaying the stored output. To generate a complete SBG, all "normal" commands offered by CyPhOS were executed. Commands that crash or reboot the system or start infinite loops were omitted, as well as the option to start two stress tests in parallel. To test the EMSComponent, the trace generator configuration used to acquire DIMs (see Section 3.4) was used. Table A.1 lists the complete sequence of steps required to recreate the SBG.

To obtain a DIM for experimentation that was robust to behavioral changes in the hardware, the EMSComponent was run 15 times following the experimental protocol outlined in Table A.2. Then a DIM for the first run was learned using $k = 10$ and all logs were tested against the DIM. The first violating log was iteratively added to the learning set until all 15 logs were accepted.

The learning set was then used to derive DIMs for $k = 1, \ldots, 10$. Section 7.1 examines the structure of the resulting set of DIMs; their suitability for use in oracles will be examined in Section 7.4.1. The model learned with $k = 1$ is rendered in Figure 6.4 using the `circo` tool from GraphViz[3] [GN00]. The transition labels use the numerical DIM event IDs.

6.4 Issues and Workarounds

While some minor bugs in CyPhOS and the EMSComponent were discovered and fixed while deploying the framework, these are of no particular interest to this thesis and are therefore omitted.

A weakness of the CyPhOS UART driver was uncovered when receiving data at the maximum baud rate of 460 800 (= 57 600 bytes per second, 3 840 event messages per second). When receiving data, the driver reads a single byte from the UART and sends an event to interested components (i.e., the monitor component). Even when immediately returning from the event handler, the event system emitted errors after a few seconds due to exhaustion of the pending event pool. This is an architectural weakness in the driver itself. To resolve it, the driver could, e.g., use a receive buffer that is passed when full. This was deemed to be outside the scope of this thesis.

To circumvent this, a virtual input was created by storing the output of a previous run of the target system in a C++ `byte[]` array. When receiving a specific command, the monitor reads input data from the array, and benchmarks the processing speed. This feature also proved to be useful for automating experiments.

The EMSComponent originally used the system timer to schedule injection and ignition starts and stops. Instrumenting the timer proved to be infeasible due to the large number of events. This resulted in inputs that were not recorded by the DIM tracer, which made inferring a DIM difficult. As a mitigation, a timerless mode was implemented that exchanged the scheduled operation for a flank-triggered one: the GPIO pins were set to high and back to low immediately. This greatly improved DIM quality.

[3]`https://www.graphviz.org/`

The delays imposed by synchronous tracing hampered the EMSComponent, causing interrupts to be lost in high-load scenarios. To avoid this, EMSBench was reconfigured using the configuration in Listing A.1 to use a slower driving cycle than that used by Schulte-Althoff. Additionally, the slow mode provided by EMSBench itself was enabled.

Chapter 7

Evaluation

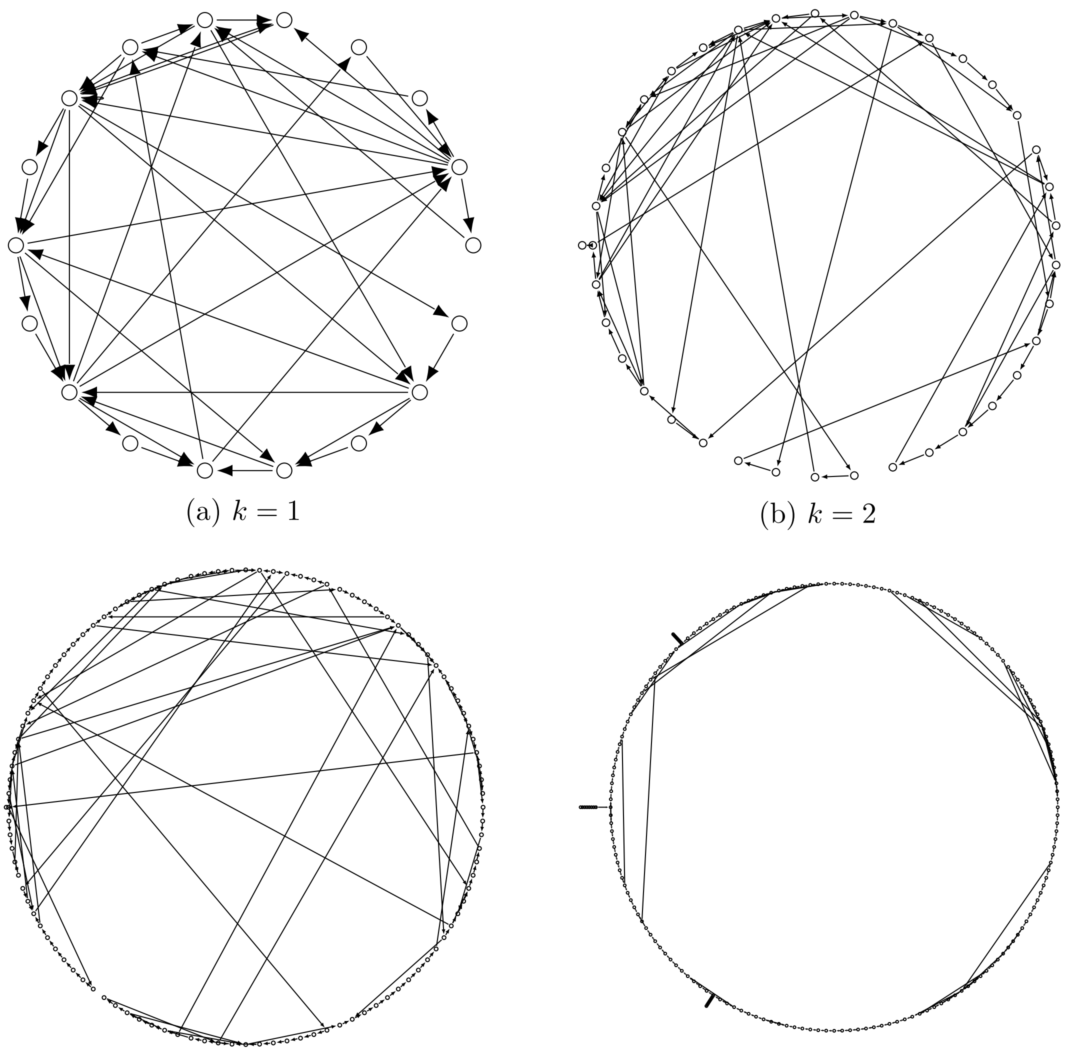
This chapter describes how the analysis framework was evaluated and the results that were obtained. First, some remarks on the learned DIMs are made. After introducing the evaluation metric and the approach to experiment design, the various parameters available to the framework are compared and an "ideal" configuration for SBFL analysis is derived. Finally, some potential threats to the validity of the results are discussed.

7.1 Informal Observations on DIMs

When visually inspecting renderings of the learned driver interaction models, several observations can be made. To create a rendering, the `circo` tool from GraphViz[1] [GN00] was used again. Other graph layout engines did not yield graphs with any discernible structure. Sample renderings are provided in Figure 7.1, a complete set of all graphs is available in Figures A.1 to A.3. As was expected, the graph's size increases in proportion to the parameter k. Of more interest is the structure of the graph. With increasing size, the outer circle of nodes with mostly sequential transitions increases in size, while the number of interconnects across the circle drops.

Since the EMS' behavior is inherently repetitive, this effect was to be expected for a valid DIM; in fact, a manually created DIM could be laid out as a circle with no interconnect, combined with a small startup sequence to account for stabilizing voltage on the GPIO when supplying power to the Discovery board. This indicates that while a larger value of k improves the fit of the model to the input data, too large values result in models "under-fitting" the acutal behavior, i.e., missing relevant information.

[1]`https://www.graphviz.org/`


(a) $k = 1$

(b) $k = 2$

(c) $k = 5$

(d) $k = 10$

Figure 7.1: Learned DIMs with increasing parameter k

The states of the DIM, however, can not be fully matched to the EMS' internal state (i.e., the internal state of the simulated motor). Further research on DIMs is suggested in Section 9.2.

7.2 Basic Concepts

To judge analysis quality, an evaluation metric is required. Since the normal SBFL workflow (see Listing 2.1) requires time-consuming manual checking of the components in its last step, a sensible metric is the amount of time saved for the human checker.

It is assumed that the checker will start with the most suspicious component and proceed to the least suspicious one. Each component that is more suspicious than the faulty one therefore wastes the checker's time. This intuition is codified in the EXAM score.

If a system with k components is analyzed using an SBFL-based fault localization scheme, let C_0 by a faulty and $C_1, \ldots, C_{k-1}$ be non-faulty components. Let $m(C_i)$ be the suspiciousness assigned to C_i. Additionally, let E_i be true if and only if C_i was executed at least once. The **EXAM score** [WDX12] for the system is

$$e = \frac{|\{i \mid E_i, m(C_i) > m(C_0), i \geq 1\}|}{|\{i \mid E_i\}|}.$$

The EXAM score is not only dependent on the SBFL scheme, but also on the test case used. Note that for randomly assigned suspiciousness, the score should approach 50 %. If multiple faults are present, the proportion of non-faulty components "above" the most suspicious faulty component is computed.

For example, if a program consists of non-faulty components A, B, C, D, E and a faulty component F. After several runs, suspiciousness is computed, with

$$\begin{aligned} m(A) &= 0.8 & m(B) &= 0.8 & m(C) &= 0.6 \\ m(D) &= 0.5 & m(E) &= 0.4 & m(F) &= 0.6. \end{aligned}$$

However, components D and E have not been executed. The EXAM score would be $\frac{2}{4} = 50\,\%$, since 4 components were executed and A and B are more suspicious than F. The identical suspiciousness of C does not reduce the score.

Since parameter classification results in multiple instances of a component being observed, one for every combination of classified parameters, the EXAM score needs to be adapted. Since the source code for all instances is identical, the checker is assumed to discover a fault during the first examination of a component. Therefore, the best score for a component is selected.

7.3 Fault Synthesis

To empirically analyze fault localization techniques, a repository of faults that can be introduced into the examined program is required. This section discusses both the basic concepts of artificial faults and the creation of the faults used for evaluation.

```
--- components/EMSMotor.cc    2018-06-19 01:40:26.986141936 +0200
+++ MLPA_158.cpp    2018-06-27 13:15:17.510079885 +0200
@@ -289,2 +289 @@
-    doWorkOfCylinder(&EMSMotorCylinder::cylinder[1]);
-    doWorkOfCylinder(&EMSMotorCylinder::cylinder[2]);
+    ;
```

Listing 7.1: “Missing small and localized part of the algorithm”-type fault generated by `clang-sfi`.

7.3.1 Basic Concepts

Faults used for this purpose can either be real faults that have been discovered in the software previously or can be artificially created. While collections of real faults such as iBugs[2] [DZ07] exist, artificial samples can be generated for any program, making them a more versatile option.

Duraes and Madeira provide a categorization of faults [DM06] that can be artificially generated that divides them into three basic categories:

- missing constructs (e.g., a missing assignment),
- wrong constructs (e.g., a wrong constant), and
- extraneous constructs (e.g., an extraneous assignment).

Inside these categories, faults are classified by the exact nature of the introduced modification. By modeling these as patches that can be applied to existing source code, a collection of faults can be synthesized for any program.

The application `clang-sfi` [Sie18] can inject several of the types of faults into C++ code. Contrary to the solution developed by Duraes and Madeira, `clang-sfi` injects faults at source code level, not into the object code generated by the compiler. An example of a generated “missing small and localized part of the algorithm”-type fault in patch format is depicted in Listing 7.1. Two lines from the motor cylinder control algorithm are removed and replaced with an empty statement.

7.3.2 Sample Creation

To obtain a set of faults, `clang-sfi` was run for each of the four C++ implementation files of the EMSComponent. In the next step, the resulting patch files were applied and a faulty program built for each.

[2] `https://www.st.cs.uni-saarland.de/ibugs/`

Unfortunately, several of the patches were defective, in that they either were empty, did not modify the source code or injected faulty syntax. These patches were discarded. All patches from the "missing return statement" family, as well as a small set of other patches, triggered compiler warnings due to execution paths without a `return` statement. Arguably, these faults were already diagnosed correctly by the `g++` C++ compiler; they were however retained.

For every type of fault a set of five faults was picked at random. Faults were discarded and replaced if

- they only modified dead code such as never-called methods or removed always-false clauses in or-statements,
- did only result in modifications to debug output,
- created a fault similar to an already used one (e.g., removed entries from the same array), or
- the resulting program was inoperable (e.g., did not boot).

A set of 41 patches remained. Some types of faults were never created or all instances had to be discarded; for other types, less than five patches could be used.

All faulty programs were deployed to the Wandboard and output recordings were created by handling a single driving cycle using the EMSComponent, using the experimental protocol described in Table A.3. These recordings were used for all subsequent experiments. The faults are identified by their type and a number.

7.4 Experiments

Having obtained a library of faults, the evaluation itself could be performed. To obtain results, the following experiments were run:

DIM Selection: This experiment selected a driver interaction model that produced no false positives, but also had a high recall, i.e., amount of faults that triggered the DIM oracle. This experiment also aimed to determine a suitable reset parameter, i.e., the number of successful transitions required after a failed one to "reset" the DIM (see Section 3.3 for details).

Method Call Sequence Length Selection: The next experiment tried to determine a suitable length parameter for MCSHS. Since longer sequences stress the CPU during analysis, timing measurements were performed to determine the highest length the Wandboard could handle.

Common Parameter Comparison: Using the results of the previous two, this experiment searched for the "ideal" configuration for thread separation,

Exp.	Oracles	Exp.	Oracles	Exp.	Oracles
MFC-1	DIM	MIES-2		MPLA-2	
MFC-2	DIM	MIES-4		MPLA-10	DIM
MFC-4		MIES-6	DIM	MPLA-26	DIM
MFC-5		MIES-7	DIM	MPLA-29	DIM
MFC-6	DIM	MIES-11	DIM	MPLA-30	DIM
MIA-2		MIFS-1		MRS-1	
MIA-3		MIFS-3		MRS-9	DIM
MIA-5		MIFS-5		MRS-10	CPU Ex. & DIM
MIA-6		MIFS-7	DIM	MRS-15	
MIA-7		MIFS-8	DIM	MRS-18	DIM
MIEB-2	DIM	MLOC-3	DIM & SBG	MVIV-7	
MIEB-3	DIM	MLOC-4	DIM	MVIV-15	
MIEB-4	DIM	MLOC-5			
MIEB-7	DIM	MLOC-8			
MIEB-10	DIM				

Table 7.1: Oracle-detected failures for the selected faults

choice of transaction detector and suspiciousness metric. All combinations were compared using the average EXAM score over all generated faults.

Evaluation of Other Parameters: Several parameters were difficult to test, for others, the first two experiments could only exclude certain options. This experiment investigated those parameters by modifying the “ideal” configuration derived in the previous step and computing the respective EXAM scores.

7.4.1 DIM Selection

In the first experiment, a suitable driver interaction model was selected. When testing the models five times in combination with a fault-free program, it was discovered that models learned using $k \geq 3$ produced false positives. Therefore, only DIMs learned using $k = 1$ and $k = 2$ were deemed suitable for use as an oracle.

When using a $k = 2$-DIM, failures were detected for 22 of the 41 faults (53.7 %), almost all by the DIM oracle (see Table 7.1 for a breakdown). The SBG oracle triggered twice, once due to a change in the control flow and once due to a system crash. In the latter case, the SBG had not “learned” about the CPU exception handling code; the CPU exception oracle was also triggered. The crash was caused by a missing return statement.

To fix the parameters of the DIM oracle, three modes of operation were compared by their failure detection rates: a $k = 1$-DIM with reset threshold 1, a $k = 2$-DIM with reset threshold 1 and the same DIM with threshold 2. Only the failures triggering the DIM oracle were examined.

As can be seen in Table 7.2, detection rates for the $k = 1$-DIM dropped to 19.5 %. The difference in the number of failures detected by the $k = 2$-DIM did not vary substantially depending on the reset threshold; only two experiments were impacted at all. The $k = 1$-DIM was subsequently discarded.

7.4.2 Limits to Method Call Sequence Lengths

In the next experiment, the suitable lengths for method call sequences were determined. It quickly became apparent that increasing the sequence length resulted in a major reduction in processing speed. As determined in Section 6.4, a processing speed of 57 600 bytes per second was essential. Since processing speed decreased for longer inputs, the recording used for SBG learning – the longest recording created during experimentation – was used for benchmarking.

Since the impact of failure indexing and thread separation had not yet been explored, all four combinations of the two parameters were tested. The results over three runs are shown in Table 7.3. Both the average speed and the speed of the lowest 10-kilobyte block were determined. The standard deviation is omitted due to being close to zero. Experiments for a length of 3 were not completed since initial attempts demonstrated an unacceptable processing speed.

Disabling the MCSHS feature resulted in a processing speed suitable for real-time processing in some configurations due to the minimum speed being above 57 600 bytes per second. Other configurations dropped below this minimum speed, but maintained a higher average speed. In these cases, the addition of a receive buffer would suffice to maintain real-time capabilities. Note that the monitor’s code contained several MCSHS-specific parts. The creation of optimized code paths for analyses without MCSHS is likely to result in a performance increase.

For sequence lengths of two, average processing speed was too slow by a factor between 2.8 and 7.7. Further examination was performed using a length of one (i.e., without MCSHS). Since optimization might conceivably increase performance to sufficient levels to handle lengths of two, the determined “ideal” configuration for sequence length one was tested for a length of two in a later step to determine the impact of sequence length on analysis quality.

Exp.	**DIM Failures**		
	$k = r = 1$	$k = 2, r = 1$	$k = r = 2$
MFS-1	0	2	2
MFC-2	13	1	1
MFC-6	1	1	1
MIEB-2	1	1	1
MIEB-3	1	1	1
MIEB-4	0	10	10
MIEB-7	0	3	3
MIEB-10	0	1	1
MIES-6	0	3	1
MIES-7	0	2	2
MIES-11	0	1	1
MIFS-7	0	6	6
MIFS-8	0	2	2
MLOC-3	11	11	11
MLOC-4	1	2	2
MLPA-10	1	1	1
MLPA-26	2	2	1
MLPA-29	0	1	1
MLPA-30	0	2	2
MVIV-9	0	10	10
MVIV-18	0	1	1
Detection Rate	19.5 %	51.2 %	51.2 %
$\sum$ Failures	31	64	61

Table 7.2: Number of failures detected by the DIM oracle for learning parameter k and reset threshold r

7.4.3 Comparison of Common Parameters

Next, an “ideal” analysis configuration was determined; however, some parameters were not investigated in this step. For all faults but two, only the DIM oracle was triggered. Since this oracle is not capable of failure indexing, this setting was ignored. The same issue arose for the reset threshold, which produces identical results for most faults.

Seq. Len.	F. Ind.	Th. Sp.	Avg. B / Sec	Min. B / Sec
2	yes	yes	11 196	5 441
2	yes	no	7 445	3 344
2	no	yes	20 788	10 032
2	no	no	12 813	5 777
1	yes	yes	*81 335*	48 819
1	yes	no	*68 147*	39 683
1	no	yes	*146 787*	*95 129*
1	no	no	*127 094*	*76 523*

Table 7.3: Processing speeds for different campaign configurations

Since the “natural” threshold for a DIM is the learning parameter k (due to the merging behavior of k-tails), a threshold of two was picked initially. Because activating failure indexing reduced performance and made scoring more complicated, it was disabled at first. The timer transaction detector was set to an interval of 100 milliseconds using injected timer events.

Three parameters remained to be fixed: the use of thread separation, the best transaction detector (timed, OSC-based, both) and the best suspiciousness metric. To determine these, a set of six experiments was run. In every experiment, the average EXAM score over all detected fault was determined for each implemented suspiciousness metrics.

The avarage EXAM score over all faults and the score’s standard deviation are tabulated in Table 7.4. Since in many experiments the fault localization was “perfect”, i.e. an EXAM score of 0 % was achieved, the number of faults handled “perfectly” is also shown.

Several conclusions can be drawn. The most obvious is the immense decrease in quality when enabling thread separation. Since the workload was mostly single-threaded, the added spectra appeared to be of little use. For parallel workloads, this effect might not persist.

The timer transaction detector outperformed the OSC-based one; disabling the latter slightly *increased* result quality. This might be caused by fact that the OSC detector cannot recognize interrupt handlers. Again, the workload might benefit the timer-based detector since the thread of execution contains pauses between OSC activations.

The high performance of the Barinel and Tarantula metrics in the last experiment gives some insight into the characteristics of these metrics; apparently, they

Th.	**Tr.**	**0 %**	**Avg.**	**σ**	**0 %**	**Avg.**	**σ**	**0 %**	**Avg.**	**σ**
Sp.	**Dt.**		BARINEL			D^2			D^3	
			Ochiai			Op2			Tarantula	
off	both	13	3.7 %	7.2 pp	15	3.0 %	5.9 pp	15	3.0 %	5.9 pp
		12	5.0 %	8.1 pp	12	5.0 %	8.1 pp	10	5.4 %	8.0 pp
off	osc	11	5.1 %	8.1 pp	13	4.7 %	8.2 pp	13	4.7 %	8.2 pp
		13	4.7 %	8.2 pp	13	4.7 %	8.2 pp	11	5.1 %	8.1 pp
off	timer	13	2.9 %	4.1 pp	*15*	*2.7 %*	*4.7 pp*	*15*	*2.7 %*	*4.7 pp*
		14	3.0 %	4.7 pp	14	3.0 %	4.7 pp	12	3.6 %	4.7 pp
on	both	4	21.7 %	19.7 pp	0	35.9 %	14.9 pp	0	36.3 %	15.1 pp
		0	36.0 %	14.8 pp	0	36.4 %	15.0 pp	4	22.5 %	19.7 pp
on	osc	1	43.2 %	21.2 pp	1	44.5 %	18.4 pp	0	43.1 %	15.3 pp
		1	44.5 %	18.4 pp	0	43.1 %	14.6 pp	1	43.2 %	21.2 pp
on	timer	13	5.6 %	10.5 pp	0	27.4 %	8.8 pp	0	27.8 %	8.8 pp
		0	27.8 %	8.7 pp	0	28.3 %	8.7 pp	12	6.0 %	10.5 pp

Table 7.4: Accuracy results for different analysis configurations

outperform the others if the number of successful runs vastly exceeds the execution counts for the examined component.

The "ideal" configuration was obtained by disabling both thread separation and the OSC transaction detector while using the D^2 metric. Since D^2 slightly outperformed D^3 in other experiments, it was given preference. While the metrics correctly identified the defective component in 15 cases, the components clustered on identical suspiciousness values, with an average of ca. 7 components tied in suspiciousness with the "correct" one. Also, it must be noted that the EXAM score obtained for all configurations without thread separation is within 1σ of the "ideal" one.

7.4.4 Examination of Edge Cases

In the final step, the features skipped in the previous experiment were examined. This was done by modifying the "ideal" configuration and recomputing the average EXAM score over all detected fault for each implemented suspiciousness metrics.

Testing failure indexing was complicated by two facts: little experimental data was available and only one fault was injected per experiment, i.e., only one "correct" location could be discovered. It was only performed for the two experiments

Mode	0 %	Avg.	σ	0 %	Avg.	σ	0 %	Avg.	σ
		BARINEL			D^2			D^3	
		Ochiai			Op2			Tarantula	
Ignored	12	2.9 %	3.8 pp	13	2.7 %	3.9 pp	13	2.7 %	3.9 pp
Parameters	12	3.0 %	4.0 pp	12	3.0 %	4.0 pp	10	3.6 %	4.0 pp
Reset	12	3.2 %	4.2 pp	14	3.0 %	4.7 pp	14	3.0 %	4.7 pp
Threshold 1	13	3.3 %	4.7 pp	14	3.0 %	4.7 pp	11	3.9 %	4.7 pp
Seq. Len. 2	12	3.8 %	5.5 pp	14	2.7 %	4.9 pp	14	2.7 %	4.9 pp
Best Score	13	2.9 %	4.9 pp	*14*	*2.4 %*	*4.6 pp*	11	4.1 %	5.5 pp
Seq. Len. 2	9	18.8 %	24.1 pp	10	17.1 %	22.2 pp	10	19.2 %	23.3 pp
Avg. Score	9	17.9 %	23.7 pp	10	21.1 %	28.0 pp	8	19.3 %	24.3 pp

Table 7.5: Accuracy results for variants of the "ideal" configuration

that had triggered multiple oracles, without significant results. Most failures were registered only once and in the same transaction, the resulting diagnosis was therefore identical to the analysis without failure indexing. One failure was attributed to a different transaction, resulting in worsened diagnostic accuracy. Without more data, a verdict on the extension is difficult, however, the feature seems to be of little benefit in conjunction with a DIM oracle.

Results for the following experiments are tabulated in Table 7.5. Disabling parameter recording was tested without measurable impact on the result accuracy. Then, a DIM reset threshold of one was tested with minor negative impact on the score.

Finally, a method call sequence length of two was tested. First, the sequences were used as-is, with the assumption of the developer examining sequences in order of suspiciousness. The best-scoring sequence containing the faulty component was chosen. In this scenario, Op2 was able to outperform D^* with a score of 2.4 %, however, the number of 0 %-results decreased. It is questionable if this offsets the performance loss. When using the averaging proposed by Dallmeier, Lindig, and Zeller, the EXAM scores dropped to values of more than 17 %.

7.5 Memory Requirements

While the Wandboard offers 2 GB of RAM, other embedded systems are equipped with substantially smaller amounts. To ensure that the framework can also operate on these systems, the memory usage after each experiment was checked. The

Mode	Avg. Usage	Usage σ
Seq. Len. 1, no threads	6 492.5 B	214.5 B
Seq. Len. 1, thread sep.	10 846.3 B	236.4 B
Seq. Len. 2, no threads	210 167.5 B	16 409.4 B

Table 7.6: Average memory usage for different thread separation modes and sequence lengths

results, using no failure indexing, are tabulated in Table 7.6. Due to the low number of faults triggering multiple oracles, no reliable results could be determined; in the two tested cases, the greatest amount required was 57 780 Bytes.

In all cases, less than one megabyte of memory was used. In the “ideal” configuration, less than 10 kilobytes were required. This amount of memory should be available in for any system equipped with sufficient processing power to perform analysis in the first place.

This does, however, not necessarily extend to MCSHS-based analysis; to make this feasible on low-memory devices, the storage system would require immense optimization. The possibility to switch data structures to achieve this was not investigated in this thesis.

7.6 Threats to Validity

The validity of these results may be threatened for two reasons. Aspects of the experimental design might prevent generalization of the results. Systemic issues related to the experimental design draw into question the validity of the results in general.

7.6.1 Experimental Design

While the EMSComponent might be representative of many low-level embedded applications, its structure is highly different from programs that might be found on a traditional PC. The low number of functions and the deterministic control flow may have reduced the effectiveness of the SBG oracle. The fact that the EMSComponent communicated with the EMS hardware without a large driver stack or buffers often cause a faults to immediately trigger the DIM oracle. This benefited the analysis process, an effect that might not be present when using a multi-layered driver stack.

Many aspects of the experimental conditions are specific to the EMSComponent, or at least to similar programs for cyber-physical systems. The low degree of parallelism is rater typical for single-purpose systems; a web server or desktop computer executes threads in parallel almost constantly.

Unfortunately, other CyPhOS applications that could be injected with meaningful faults were not readily available. While the effort required to port the framework to another platform is low, this was nonetheless deemed to be outside of the scope of this thesis. Whether this thesis' results can be generalized to other systems remains to be explored.

7.6.2 Systemic Issues

In a review of existing results comparing suspiciousness metrics by their performance [Pea+17; Pea+16], Pearson et al. noted that these results were acquired in different environments and by using artificially generated faults. To check applicability for the "real world", the experiments were repeated while substituting real faults that were acquired from the tested programs's history for artificial faults.

While the authors were able to replicate most experimental results with sufficient degrees of significance (except for a result ranking the Barinel metric above the Ochiai metric) using artificial faults, all results were either diminished to insignificant differences between metrics or refuted outright when substituting real faults.

Since for this thesis, the injected faults were artificially generated, the validity of the obtained results may be threatened. While Pearson et al. recommend the use of real faults, a review of the CyPhOS source repository did not reveal easily-accessible faults in single components that could be reintroduced. The idea of manually creating bugs was rejected due to the risk of involuntarily introducing bias.

To improve the results obtained using artificial faults, all metrics examined by Pearson et al. were evaluated to obtain a broad set of results. Again, a port to other systems could bring clarity by making available other sources of faults.

Chapter 8

Related Work

This chapter discusses alternative strategies to tackle this thesis' objectives that were not pursued. A short overview, as well as the reasons for not adopting it, are given for each of the approaches.

8.1 Model-based Diagnosis

In the scope of the Trader project [Mat09], an alternative to the use of suspiciousness metrics was examined by Abreu, Zoeteweij, and Gemund. Using a spectrum from a failed run, it can be inferred that at least one component present in the trace must be faulty.

Instead of examining the similarity of component vectors to the error vector, this information is used in a first-order logic model [AZG08; HM09]. This model assigns a **health variable** h_i to every component C_i such that

$$h_i = 0 \iff \text{component } C_i \text{ is faulty}$$

For a failing run i executing components $C_{i_0}, \ldots, C_{i_{\max}}$, it is known that one component cannot be healthy:

$$\neg h_{i_0} \vee \neg h_{i_1} \vee \cdots \vee \neg h_{i_{\max}}$$

Using spectra from multiple failing runs $0, \ldots, n-1$, this fact must hold for every run:

$$(\neg h_{0_0} \vee \cdots \vee \neg h_{0_{\max}}) \wedge (\neg h_{1_0} \vee \cdots \vee \neg h_{1_{\max}}) \wedge \cdots \wedge (\neg h_{n-1_0} \vee \cdots \vee \neg h_{n-1_{\max}})$$

A fault localization diagnosis can be acquired by finding a variable assignment that only assigns a false value to a minimal number of health variables. Computing a minimal assignment is NP-complete (hitting set problem). Instead of using an exponential-time algorithm to identify a minimal assignment, the Staccato algorithm [AG09] was devised. It uses regular SBFL suspiciousness results to assign an initial non-boolean value to each health variable and then refines these values to yield hitting sets.

Since the generation on the models requires information about every individual run, instead of aggregate matrices, every spectrum must be stored. Due to the memory requirements, this approach was deemed unsuitable for implementation in an embedded context and not pursued in this thesis.

8.2 Optimal Suspiciousness Metrics

To improve upon the empirical evaluation of suspiciousness metrics, Xie et al. presented a comparison relation between metrics [Xie+13]. Using this relation, two equivalent, optimal metrics among a set of thirty were identified. However, this ranking only holds under the assumption that 100 % test coverage is available and does not necessarily extend to identifying missing statements.

Le, Thung, and Lo examined these optimal metrics in real-world scenarios that did not match the assumptions required for the theoretical comparison [LTL13]. In this scenario, the optimal metrics were outperformed by non-optimal ones.

This thesis uses synthetic bugs that do remove statements and forsakes test coverage for other approaches. Therefore, only popular, non-optimal metrics that proved accurate in empirical studies were examined.

8.3 Fault Localization in Embedded Devices

For a different part of the Trader project [Mat09], SBFL was used by Abreu et al. to investigate pre-existing faults in a TV set's controller software [Abr+09; Zoe+09]. This environment severely limited the available memory and CPU time.

To successfully employ SBFL, several optimizations were implemented. Spectra were recorded into a fixed-size ring buffer fur further processing during periods of low system load and discarded if necessary. The spectra were then integrated into aggregate matrices to reduce memory consumption further. This approach maintained the timing characteristics of the tested system sufficiently to correctly diagnose a timing-related fault. Even with a little collected data, good fault localization results could be obtained for the investigated bugs.

This thesis uses similar approaches to reduce memory consumption on the monitor. Since the framework developed for this thesis does not need to contest the target's CPU, a buffered approach was not deemed necessary.

8.4 Energy Consumption Models

To formalize the power consumption of wireless sensor networks, Wang and Yang proposed **communication subsystem energy consumption models** [WY07] that model the power states of sensors (off, power save, receiving, transmitting, etc.) as a finite state machine.

This approach was extended by Falkenberg for use in driver design [Fal14]. Since in a low-power environment, the OS needs to enforce a regimen on the power consumption of peripheral components to avoid a voltage drop, drivers should be aware of the power consumption state of the controlled devices. This was realized by modeling these states in the drivers as priced timed automatons.

Friesel investigated the feasibility of automatic refinement of these models [Fri17], concluding that such refinement could reliably improve the accuracy of the models. Since otherwise, these models would have required manual refinement, the approach has the potential to reduce the workload for human operators.

While these approaches concentrate on modeling power consumption with high accuracy, the DIMs examined in this thesis are designed to detect failures of the driver itself. Therefore, the models need to be less accurate, but need to avoid false reporting of failures.

For the energy consumption models, the initial models were designed manually, using a set of states designed to model the actual states of the hardware. DIMs' states are inferred using a fully automatic process; their states do *not* need to model a property of the hardware.

8.5 Learning Cache Replacement Policies

To reverse-engineer the cache replacement policy of a CPU, Rueda Cebollero used LearnLib[1] to infer register mealy automata from the timing behavior of cached memory [Rue13].

[1] `https://learnlib.de/`

For a simulated CPU, an accurate model was obtained. When using data acquired from measurements, severe limitations of the approach – primarily due to performance problems – became apparent, leading to the conclusion that future work would be required to make the approach feasible.

While this work attempts to model the hardware, driver interaction models describe the communication between driver and hardware. The performance issues could be avoided in this thesis by using a different learning approach for DIMs.

Chapter 9

Conclusion

Drawing on the results presented in previous chapters, this chapter discusses the conclusions that can be drawn from the experiments and suggests future work building on this thesis.

9.1 Results

The goals outlined in Section 1.1 have been fully achieved. The designed framework meets the constraints and the evaluation results build confidence that the system can be a useful aid in debugging. The performance constraints of embedded systems were met, although the use of UART as a communication medium caused several complications.

The performance of domain-agnostic oracles and transaction detectors was varied. While the performance of the SBG oracle was disappointing, the DIM-based oracle successfully detected over 50 % of injected faults. The timer-based transaction detector was able to outperform the manually written one. It can be concluded that the approach of obtaining these components using machine learning is viable for fault localization. Even outside of SBFL, DIMs might be of utility in the field of model checking.

The accuracy of the SBFL process was beyond expectations. While literature reports average EXAM scores of 10 to 20 %, the 2.7 % achieved in the “ideal” configuration are a vast improvement over these values. Should this quality generalize to other embedded systems, the analysis framework can successfully compete with other fault localization techniques.

The extensions to SBFL – aside from parameter classification – that were examined yielded disappointing results. While positive effects might appear for more complex, larger systems, simplicity seems to be its own reward for the embedded case. Skepticism about the suitability of MCSHS for this domain appears to be warranted, since the (questionable) improvement in quality is bought with severe performance reductions.

All in all, SBFL was confirmed to be a viable, useful tool in localizing faults. It can only be hoped that the technology rises to more prominence among software developers in the years to come.

9.2 Future Work

Many properties of the framework had to remain unexplored. Future research is required to generalize the thesis' results to other applications and platforms or demonstrate the inability to do so. On the one hand, performance on other embedded systems would be of interest to study the usability in a general embedded context. On the other hand, examining desktop applications might give insight into the suitability of models such as SBGs in these scenarios.

Due to the pluggable nature of oracles, transaction detectors and suspiciousness metrics, the framework can be used as an experimental platform for these. Of special interest might be the derivation of other domain-agnostic components and their evaluation in different contexts. Finally, the framework implementation certainly leaves room for performance optimization. Performance improvements might make MCSHS viable or allow porting the monitor to even lower-power platforms.

Driver interaction models could only be explored in passing. While model checking often has focused on modeling a single component (i.e. *either* the program *or* the hardware), the holistic approach of DIMs and similar interaction models might help shed light on previously unexplored aspects of system behavior.

If the "readability" of models can be improved, they might prove to be of interest as a reverse engineering tool. Having a model of the observed communications might aid in the understanding of undocumented network protocols or device drivers.

The models themselves might benefit from a different machine model, such as timed or register automata. To improve the recognition of abnormal behavior, probabilistic automata could be used to detect shifts in the probability distribution and acquire resistance to the occasional exceptional event. Weighting edges can also improve the visualization by highlighting the "common case".

While this thesis has hopefully both advanced the understanding of SBFL in the context of embedded systems and uncovered a topic of future interest by examining DIMs, research in these fields is far from complete. Thus, this thesis concludes in the hope that future advances in those fields might eventually contribute to the understanding of software and – perhaps – eliminate a few bugs on the way.

Glossary

advice logic that needs to be executed at one or more join points to realize an aspect **24**, 25, 71

aggregate matrix a matrix recording the number of failing and succeeding runs during which a component was executed or not executed **9**, 10, 27, 31, 36, 37, 64

aspect a part of a system's function that can not cleanly be encapsulated into a component **23**, 24, 25, 27–30, 44, 71, 72

aspect weaver a component that modifies a program so that at defined pointcuts, an advice is introduced 10, **24**, 27

behavior a system's actual function **5**, 72

classification categorization of a function's parameters into subsets of their types **12**, 29–31, 36, 51, 67

classifier a function mapping all values of a type to a limited numerical range 11, **12**, 36, 38, 72

component a constituent part of a system **5**, 6–13, 15, 21, 23, 28, 29, 31, 36, 41, 44, 50, 51, 58, 59, 61, 63, 71–73

component vector a vector recording in which runs a component was executed **7**, 8, 11, 63, 73

driver interaction model a FSM modelling the interactions between driver and hardware 4, 15, **16**, 18, 49, 54, 66, 68

error vector a vector recording which runs have failed **7**, 8, 11, 63, 72, 73

EXAM score the proportion of components more suspicious than the faulty one **51**, 54, 57–59, 67

execution the part of a run affecting a single component **6**, 7, 10, 41, 51, 58, 63, 71, 73

failing exhibiting a failure **6**, 7, 8, 10, 14, 17, 31, 63, 71

failure deviation of a system's service from the intended function **6**, 7, 9, 10, 12, 13, 15, 17, 21, 27, 31, 32, 36, 44, 54–56, 59, 65, 71–73

failure indexing separate SBFL analysis for every different failure detected **21**, 32, 36, 55–60

fault the (hypothesized) cause of a failure **6**, 9–12, 14, 16, 18, 21, 27, 28, 31, 43, 44, 51–54, 56–58, 60, 61, 64, 67, 68, 72, 73

fault localization the attempt to find the fault causing a failure 4, **6**, 8–11, 14, 28, 51, 57, 63, 64, 67, 72, 87

faulty containing a fault **6**, 7, 8, 12, 15, 51–53, 59, 63, 71

function a system's intended operation **5**, 14, 23, 71, 72

join point the parts of a program affected by an aspect **23**, 24, 25, 29, 71, 72

k-**tail** the set of words of length k that are accepted starting from a state **18**, 19, 20

method call sequence a tuple containing the last k observed method calls **10**, 11, 12, 31, 36, 39, 72

method call sequence hit spectrum a spectrum using method call sequences as components 3, **10**

monitor the system analyzing the target to perform fault localization **27**, 28, 29, 33, 34, 44, 45, 65, 68

observation matrix a matrix containing a set of spectra and the corresponding error vector side by side **7**

oracle a function that detects failures in a trace 3, 4, **13**, 14, 15, 17, 21, 27, 28, 31, 32, 36, 44–46, 53–56, 60, 67, 68

pointcut a set of join points **23**, 24, 25, 34, 71

program a system implemented in software **6**, 7, 9, 10, 12, 14–18, 21, 23–25, 27, 28, 36, 45, 51–54, 60, 61, 71–73

representative a value output by a classifier, i.e., in its image **12**, 37–39

run the execution of a complete program **6**, 7–10, 13, 14, 17, 18, 31, 45, 51, 63, 64, 71, 73

service the part of a system's behavior percieved by its users **5**, 6, 72

software behavior graph a graph storing observed calls between functions and methods **14**, 15

spectrum a vector recording which components were executed during a program's run **7**, 9, 10, 12–14, 21, 31, 32, 36, 57, 63, 64, 72

succeeding exhibiting no failures **6**, 7, 8, 17, 31, 32, 71

suspiciousness the similarity of a component vector to the error vector **8**, 9, 10, 12, 32, 51, 58, 64, 71

suspiciousness metric a function computing a component's suspiciousness **8**, 9, 11, 28, 32, 54, 57, 58, 61, 63, 64, 68

system an entity that interacts with other entities **5**, 6, 12, 14, 15, 21, 23, 34, 51, 61, 64, 71, 72

target the system suspected of harboring faults 18, **27**, 28, 29, 33–35, 37, 45, 65, 72

thread separation separation of gathered data by originating thread of execution **12**, 21, 29, 31, 32, 53, 55, 57, 58, 60

trace a recording of the components executed during a program's run, in chronological order **6**, 7, 10, 13, 17, 21, 27, 28, 37, 63, 72, 73

transaction a continuous subsequence of a trace **13**, 14, 21, 27, 31, 32, 34, 45, 59, 73

transaction detector a function that splits a trace into transactions 3, 4, **13**, 14, 21, 27, 28, 31, 32, 34, 44, 45, 54, 57, 58, 67, 68

Acronyms

AOP aspect-oriented programming 3, 4, **23**, 24

API application programming interface 39

ASCII American Standard Code for Information Interchange 35

avg. average 57–60

B bytes 57

CPU central processing unit 12, 18, 29, 42, 44, 53–55, 64–66

CRC cyclic redundancy check 35

DIM driver interaction model **16**, 17–20, 23, 29, 31–33, 35, 36, 45, 46, 49, 50, 53–57, 59, 60, 65–68, 85–87, 89–91

EMS engine management system **42**, 43, 49, 50, 60

ex. exception 54

exp. experiment 54, 56

f. ind. failure indexing 57

FSM finite state machine 16–20, 65, 71

GPIO general-purpose input / output 33, 43, 49

I / O input / output 29

ID identifier 33, 46

IP Internet Protocol 16

JPID join point identifier **25**, 30, 38

MCSHS method call sequence hit spectrum **10**, 11, 12, 31, 36, 53, 55, 60, 67, 68

min. minimum 57

OS operating system 4, 27–29, 32–35, 39, 41, 65, 87

OSC operating system component **41**, 44, 45, 57, 58

PC personal computer 18, 45, 60

PCI Peripheral Component Interconnect 16

pp percentage points 58, 59

RAM random access memory 41, 42, 59

SBFL spectrum-based fault localization 3–5, **7**, 9, 10, 13, 14, 21, 23, 27–30, 32, 33, 49–51, 64, 67, 68, 72

SBG software behavior graph **14**, 15, 17, 23, 29, 32, 36, 45, 54, 55, 60, 67, 68, 86

sec seconds 57

seq. len. sequence length 57, 59

SQL Structured Query Language 14

TCP Transmission Control Protocol 16

th. sp. thread separation 57, 58

tr. dt. transaction detector 58

UART universal asynchronous receiver-transmitter 29, 32–36, 43, 46, 67

XML Extensible Markup Language 25

Bibliography

[Abr+09] Rui Abreu et al. "A practical evaluation of spectrum-based fault localization". In: *Journal of Systems and Software* 82.11 (Nov. 2009), pp. 1780–1792. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.06.035 (cit. on pp. 9, 64).

[AG09] Rui Abreu and Arjan J. C. van Gemund. "A Statistics-directed Minimal Hitting Set Algorithm". In: *Proceedings of the 20th International Workshop on Principles of Diagnosis*. Ed. by Erik Frisk et al. June 2009, pp. 51–58. URL: http://photon.isy.liu.se/dx09/ (cit. on p. 64).

[Avi+04] Algirdas Avižienis et al. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2 (cit. on pp. 5, 6).

[AZG06] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. *Program Spectra Analysis in Embedded Software: A Case Study.* Tech. rep. TUD-SERG-2006-007. Software Engineering Research Group, Delft University of Technology, 2006. arXiv: cs/0607116 (cit. on p. 21).

[AZG08] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. "An Observation-based Model for Fault Localization". In: *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008).* New York, NY, USA: ACM, July 21, 2008, pp. 64–70. ISBN: 978-1-60558-054-8. DOI: 10.1145/1401827.1401841 (cit. on pp. 7, 63).

[AZG09] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. "Spectrum-Based Multiple Fault Localization". In: *2009 IEEE/ACM International Conference on Automated Software Engineering.* IEEE, Nov. 2009, pp. 88–99. ISBN: 978-1-4244-5259-0. DOI: 10.1109/ASE.2009.25 (cit. on p. 8).

[BF72] A. W. Biermann and J. A. Feldman. “On the Synthesis of Finite-State Machines from Samples of Their Behavior”. In: *IEEE Transactions on Computers* C-21.6 (June 1972), pp. 592–597. ISSN: 0018-9340. DOI: `10.1109/TC.1972.5009015` (cit. on p. 18).

[Bri59] Rene de la Briandais. “File Searching Using Variable Length Keys”. In: *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*. New York, NY, USA: ACM, Mar. 3, 1959, pp. 295–298. DOI: `10.1145/1457838.1457895` (cit. on p. 37).

[BS15] Hendrik Borghorst and Olaf Spinczyk. “Increasing the Predictability of Modern COTS Hardware through Cache-Aware OS-Design”. In: *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. Ed. by Björn B. Brandenburg and Robert Kaiser. MPI-SWS, July 7, 2015, pp. 41–44. URL: `https://people.mpi-sws.org/~bbb/events/ospert15/` (cit. on p. 41).

[Cas+11] Paulo Casanova et al. “Architecture-Based Run-Time Fault Diagnosis”. In: *Software Architecture*. Ed. by Ivica Crnkovic, Volker Gruhn, and Matthias Book. Lecture Notes in Computer Science 6903. Berlin, Heidelberg: Springer, 2011, pp. 261–277. ISBN: 978-3-642-23798-0. DOI: `10.1007/978-3-642-23798-0_29` (cit. on pp. 13, 14).

[Cer69] Vint Cerf. *ASCII format for Network Interchange*. STD 80. RFC Editor, Oct. 16, 1969. URL: `https://tools.ietf.org/html/std80` (cit. on p. 35).

[DLZ05] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. “Lightweight Defect Localization for Java”. In: *ECOOP 2005 - Object-Oriented Programming*. Ed. by Andrew P. Black. Lecture Notes in Computer Science 3586. Berlin, Heidelberg: Springer, 2005, pp. 528–550. ISBN: 978-3-540-31725-8. DOI: `10.1007/11531142_23` (cit. on pp. 6, 10, 11, 59).

[DM06] Joao A. Duraes and Henrique S. Madeira. “Emulation of Software Faults: A Field Data Study and a Practical Approach”. In: *IEEE Transactions on Software Engineering* 32.11 (Nov. 20, 2006), pp. 849–867. ISSN: 0098-5589. DOI: `10.1109/TSE.2006.113` (cit. on p. 52).

[DZ07] Valentin Dallmeier and Thomas Zimmermann. “Extraction of Bug Localization Benchmarks from History”. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, Nov. 5, 2007, pp. 433–436. ISBN: 978-1-59593-882-4. DOI: `10.1145/1321631.1321702` (cit. on p. 52).

[Fal14] Robert Falkenberg. “Entwurf eines energiegewahren Treibermodells für eingebettete Betriebssysteme”. MA thesis. Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl 12, Arbeitsgruppe ESS, Dec. 2014. DOI: `10.17877/DE290R-16451` (cit. on p. 65).

[FBS17] Daniel Friesel, Markus Buschhoff, and Olaf Spinczyk. “Annotations in Operating Systems with Custom AspectC++ Attributes”. In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. New York, NY, USA: ACM, Oct. 28, 2017, pp. 36–42. ISBN: 978-1-4503-5153-9. DOI: `10.1145/3144555.3144561` (cit. on p. 24).

[Fre60] Edward Fredkin. “Trie Memory”. In: *Communications of the ACM* 3.9 (Sept. 1, 1960), pp. 490–499. ISSN: 0001-0782. DOI: `10.1145/367390.367400` (cit. on p. 37).

[Fri17] Daniel Friesel. “Automatisierte Verfeinerung von Energiemodellen für eingebettete Systeme”. MA thesis. Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl 12, Arbeitsgruppe ESS, Mar. 2017. DOI: `10.17877/DE290R-18206` (cit. on p. 65).

[GN00] Emden R. Gansner and Stephen C. North. “An open graph visualization system and its applications to software engineering”. In: *Software: Practice and Experience* 30.11 (Aug. 24, 2000): *Special Issue: Discrete algorithm engineering*, pp. 1203–1233. DOI: `10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N` (cit. on pp. 46, 49).

[HBL75] Joseph L. Hammond, James E. Brown, and Shyan-Shiang S. Liu. *Development of a Transmission Error Model and an Error Control Model*. Tech. rep. RADC-TR-75-138. Griffiss AFB, NY, USA: Rome Air Development Center, May 1975. URL: `https://ntrl.ntis.gov/NTRL/dashboard/searchResults/titleDetail/ADA013939.xhtml` (cit. on p. 35).

[HM09] Jozef Hooman and Somayeh Malakuti. “Model-based error detection”. In: *Trader: Reliability of high-volume consumer products. A collaborative research project on the reliability of complex embedded devices.* Ed. by Roland Mathijssen. Eindhoven, the Netherlands: Embedded Systems Institute, Oct. 17, 2009. Chap. 8, pp. 93–101. ISBN: 978-90-78679-04-2. URL: `http://redesign.esi.nl/research/applied-research/finished-projects/trader/` (cit. on p. 63).

[JH05] James A. Jones and Mary Jean Harrold. “Empirical Evaluation of the Tarantula Automatic Fault-localization Technique”. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, Nov. 7, 2005, pp. 273–282. ISBN: 1-58113-993-4. DOI: `10.1145/1101908.1101949` (cit. on p. 8).

[JHS02] James A. Jones, Mary Jean Harrold, and Sohn Stasko. "Visualization of Test Information to Assist Fault Localization". In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* IEEE, May 19, 2002, pp. 467–477. ISBN: 1-58113-472-X. DOI: `10.1145/581396.581397` (cit. on p. 10).

[Kic+01] Gregor Kiczales et al. "An Overview of AspectJ". In: *ECOOP 2001 — Object-Oriented Programming.* Ed. by Jørgen Lindskov Knudsen. Lecture Notes in Computer Science 2072. Berlin, Heidelberg: Springer, 2001. ISBN: 978-3-540-45337-6. DOI: `10.1007/3-540-45337-7_18` (cit. on pp. 23, 24).

[Kic+97] Gregor Kiczales et al. "Aspect-oriented programming". In: *ECOOP'97 — Object-Oriented Programming.* Ed. by Mehmet Akşit and Satoshi Matsuoka. Lecture Notes in Computer Science 1241. Berlin, Heidelberg: Springer, 1997, pp. 220–242. ISBN: 978-3-540-69127-3. DOI: `10.1007/BFb0053381` (cit. on pp. 23, 24).

[KU15] Florian Kluge and Theo Ungerer. "EMSBench: Benchmark und Testumgebung für reaktive Systeme". In: *Betriebssysteme und Echtzeit.* Ed. by Wolfgang A. Halang and Olaf Spinczyk. Informatik aktuell. Berlin, Heidelberg: Springer, 2015, pp. 11–20. ISBN: 978-3-662-48611-5. DOI: `10.1007/978-3-662-48611-5_2` (cit. on p. 43).

[Liu+05] Chao Liu et al. "Mining Behavior Graphs for "Backtrace" of Non-crashing Bugs". In: *Proceedings of the 2005 SIAM International Conference on Data Mining.* Society for Industrial and Applied Mathematics, 2005, pp. 286–297. ISBN: 978-0-89871-593-4. DOI: `10.1137/1.9781611972757.26` (cit. on p. 14).

[LTL13] Tien-Duy B. Le, Ferdian Thung, and David Lo. "Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization". In: *2013 IEEE International Conference on Software Maintenance.* IEEE, Sept. 2013, pp. 380–383. ISBN: 978-0-7695-4981-1. DOI: `10.1109/ICSM.2013.52` (cit. on p. 64).

[Mat09] Roland Mathijssen, ed. *Trader: Reliability of high-volume consumer products. A collaborative research project on the reliability of complex embedded devices.* Eindhoven, the Netherlands: Embedded Systems Institute, Oct. 17, 2009. ISBN: 978-90-78679-04-2. URL: `http://redesign.esi.nl/research/applied-research/finished-projects/trader/` (cit. on pp. 63, 64).

[Men42] Luigi Federico Menabrea. "Sketch of The Analytical Engine Invented by Charles Babbage". Trans. from the Italian by Augusta Ada King-Noel, Countess of Lovelace. In: *Bibliothèque universelle de Genève* 82 (Oct. 1842) (cit. on p. 1).

[Mur95] Kevin P. Murphy. *Passively Learning Finite Automata.* Working Papers 96-04-017. Santa Fe Institute, Nov. 16, 1995. URL: `https://www.santafe.edu/research/results/working-papers/passively-learning-finite-automata` (cit. on p. 18).

[NLR11] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. "A Model for Spectra-based Software Diagnosis". In: *ACM Transactions on Software Engineering and Methodology* 20.3 (Aug. 2011), 11:1–11:32. ISSN: 1049-331X. DOI: `10.1145/2000791.2000795` (cit. on p. 8).

[Och57] Akira Ochiai. "Zoogeographical Studies on the Soleoid Fishes Found in Japan and Its Neighbouring Regions—II". In: *Bulletin of the Japanese Society for the Science of Fish* 22.9 (1957), pp. 526–530. ISSN: 0021-5392. DOI: `10.2331/suisan.22.526` (cit. on p. 8).

[PB61] W. W. Peterson and D. T. Brown. "Cyclic Codes for Error Detection". In: *Proceedings of the IRE* 49.1 (Jan. 1961), pp. 228–235. ISSN: 0096-8390. DOI: `10.1109/JRPROC.1961.287814` (cit. on p. 35).

[Pea+16] Spencer Pearson et al. *Evaluating & improving fault localization techniques.* Tech. rep. UW-CSE-16-08-03. Seattle, WA, USA: University of Washington Department of Computer Science and Engineering, Sept. 2016. URL: `https://homes.cs.washington.edu/~mernst/pubs/fault-localization-tr160803-abstract.html` (cit. on p. 61).

[Pea+17] Spencer Pearson et al. "Evaluating and Improving Fault Localization". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, May 2017, pp. 609–620. ISBN: 978-1-5386-3868-2. DOI: `10.1109/ICSE.2017.62` (cit. on p. 61).

[Rep+97] Thomas Reps et al. "The use of program profiling for software maintenance with applications to the year 2000 problem". In: *Software Engineering — ESEC/FSE'97.* Ed. by Mehdi Jazayeri and Helmut Schauer. Lecture Notes in Computer Science 1301. Berlin, Heidelberg: Springer, 1997, pp. 432–449. ISBN: 978-3-540-69592-9. DOI: `10.1007/3-540-63531-9_29` (cit. on p. 7).

[RSB05] Harald Raffelt, Bernhard Steffen, and Therese Berg. "LearnLib: A Library for Automata Learning and Experimentation". In: *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems.* New York, NY, USA: ACM, Sept. 5, 2005, pp. 62–71. ISBN: 1-59593-148-1. DOI: `10.1145/1081180.1081189` (cit. on p. 16).

[Rue13] Guillem Rueda Cebollero. "Learning Cache Replacement Policies using Register Automata". MA thesis. Uppsala University, Department of Information Technology, Dec. 2013. URL: `http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A678847&dswid=5885` (cit. on p. 65).

[Sch17] Thomas Schulte-Althoff. “Validierung des Echtzeitverhaltens des ereignisbasierten Betriebssystems CyPhOS am Beispiel einer Motorsteuerung”. BA thesis. Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl 12, Arbeitsgruppe ESS, Nov. 2, 2017. URL: `https://ess.cs.tu-dortmund.de/Teaching/Theses/` (cit. on pp. 42–44, 47).

[SGS02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. “AspectC++: An Aspect-oriented Extension to the C++ Programming Language”. In: *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*. Ed. by James Noble and John Potter. CRPIT. Sydney, Australia: ACS, 2002, pp. 53–60. URL: `http://crpit.com/abstracts/CRPITV10Spinczyk.html` (cit. on p. 24).

[Sie18] Daniel Ferdinand Siegert. “Entwicklung eines Werkzeugs zur Injektion von Softwarefehlern basierend auf Clang”. BA thesis. Technische Universität Dortmund, Fakultät für Informatik, Lehrstuhl 12, Arbeitsgruppe ESS, May 28, 2018. URL: `https://ess.cs.tu-dortmund.de/Teaching/Theses/` (cit. on p. 52).

[WDX12] W. Eric Wong, Vidroha Debroy, and Dianxiang Xu. “Towards Better Fault Localization: A Crosstab-Based Statistical Approach”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.3 (May 2012), pp. 378–396. ISSN: 1094-6977. DOI: `10.1109/TSMCC.2011.2118751` (cit. on p. 51).

[Won+14] W. Eric Wong et al. “The DStar Method for Effective Software Fault Localization”. In: *IEEE Transactions on Reliability* 63.1 (Mar. 2014), pp. 290–308. ISSN: 0018-9529. DOI: `10.1109/TR.2013.2285319` (cit. on p. 8).

[Won+16] W. Eric Wong et al. “A Survey on Software Fault Localization”. In: *IEEE Transactions on Software Engineering* 42.8 (Aug. 1, 2016), pp. 707–740. ISSN: 0098-5589. DOI: `10.1109/TSE.2016.2521368` (cit. on p. 10).

[WY07] Qin Wang and Woodward Yang. “Energy Consumption Model for Power Management in Wireless Sensor Networks”. In: *2007 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*. IEEE, June 2007, pp. 142–151. ISBN: 1-4244-1268-4. DOI: `10.1109/SAHCN.2007.4292826` (cit. on p. 65).

[Xie+13] Xiaoyuan Xie et al. “A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization”. In: *ACM Transactions on Software Engineering and Methodology* 22.4 (Oct. 1, 2013), 31:1–31:40. ISSN: 1049-331X. DOI: 10.1145/2522920.2522924 (cit. on pp. 8, 64).

[Zoe+09] Peter Zoeteweij et al. “Spectrum-based fault localization in practice”. In: *Trader: Reliability of high-volume consumer products. A collaborative research project on the reliability of complex embedded devices.* Ed. by Roland Mathijssen. Eindhoven, the Netherlands: Embedded Systems Institute, Oct. 17, 2009. Chap. 10, pp. 113–124. ISBN: 978-90-78679-04-2. URL: http://redesign.esi.nl/research/applied-research/finished-projects/trader/ (cit. on p. 64).

Appendix

The appendix contains experimental protocols, configuration files and renderings of all driver interaction models that might be helpful for future work. These are rendered on the following pages.

Step	Action Performed
1	build the EMSBench trace generator using the DIM car profile and driving cycle (see Listing A.1) and deploy it on the Discovery board
2	build the target and deploy it on the Wandboard
3	power off and disconnect the Discovery board
4	power cycle the Wandboard and start a new recording
5	connect the Discovery board
6	send a motor start command `m`
7	power on the Discovery board
8	wait for the end of the simulated driving cycle
9	send a motor stop command `moff`
10	send a stress test start command `son`
11	wait for ca. 10 seconds
12	send a stress test stop command `soff`
13	send a non-existing command `x`
14	send a command `help`
15	send a command `eventstats`
16	send a command `cachestats`
17	send a command `pmustats`
18	send a command `uptime`
19	send a command `version`
20	send a command `verify`
21	send a command `misstest`
22	send a command `tlbinfo`
23	send a command `memtest`
24	send a command `temps`
25	send a command `benchmark`
26	end recording
27	build the monitor with the recording as in-memory input and deploy it on the Wandboard
28	power cycle the Wandboard and start a new recording
29	send a binary input-from-memory command
30	send a binary print-learned-SBG command
31	end recording

Table A.1: Experimental protocol used to obtain an SBG for CyPhOS

Step	Action Performed
1	build the EMSBench trace generator using the DIM car profile and driving cycle (see Listing A.1) and deploy it on the Discovery board
2	build the target and deploy it on the Wandboard
3	power off and disconnect the Discovery board
4	power cycle the Wandboard and start a new recording
5	connect the Discovery board
6	send a motor start command `m`
7	power on the Discovery board
8	wait for the end of the simulated driving cycle
9	send a motor stop command `moff`
10	end recording

Table A.2: Experimental protocol used to obtain a recording for constructing DIMs

Step	Action Performed
1	build the EMSBench trace generator using the DIM car profile and driving cycle (see Listing A.1) and deploy it on the Discovery board
2	build the target and deploy it on the Wandboard
3	power off and disconnect the Discovery board
4	power cycle the Wandboard and start a new recording
5	connect the Discovery board
6	send a motor start command `m`
7	power on the Discovery board
8	wait for the end of the simulated driving cycle
9	should the OS not have crashed, send a motor stop command `moff`
10	end recording

Table A.3: Experimental protocol used to obtain a recording for fault localization

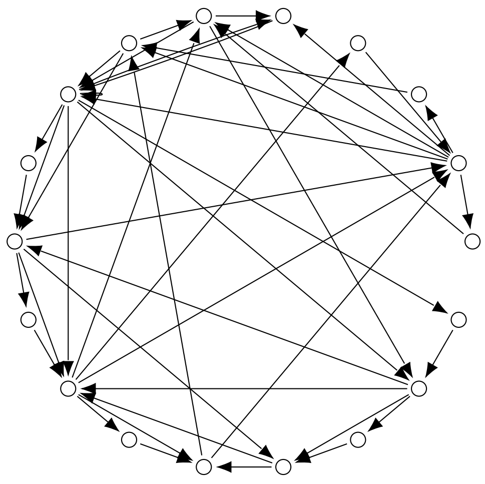
```
# Tyre
width = 175
aspect_ratio = 65
diameter = 15

# Engine and car
idle_rpm = 50
# Acceleration to idle s^{-2}
acc_to_idle = 20

# Gear translation
gear[1] = 3.545
gear[2] = 1.913
gear[3] = 1.31
gear[4] = 1.027
gear[5] = 0.85
gear[6] = 0

# Axle and cardan translation.
axle = 4.294

primary_teeth = 12

# offset of secondary tooth
offset_secondary = 0.04
```

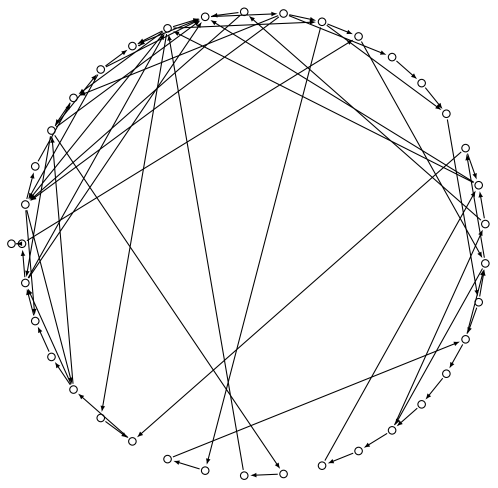
(a) car configuration

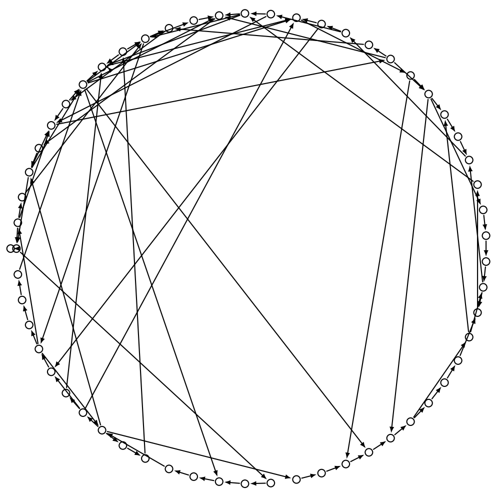
```
# Acc. ; SpeedS ; SpeedE ; Dur. ; Gear
       ;      0 ;     15 ;    1 ;    5
       ;     15 ;     15 ;   30 ;    5
       ;     15 ;      0 ;    1 ;    5
```

(b) driving cycle

Listing A.1: Car and driving cycle configuration for EMSBench


(a) $k = 1$


(b) $k = 2$


(c) $k = 3$

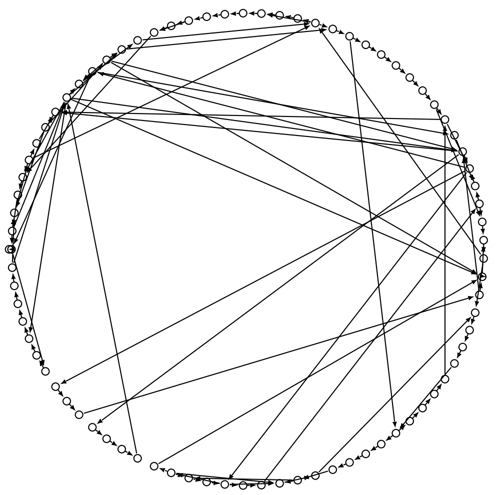
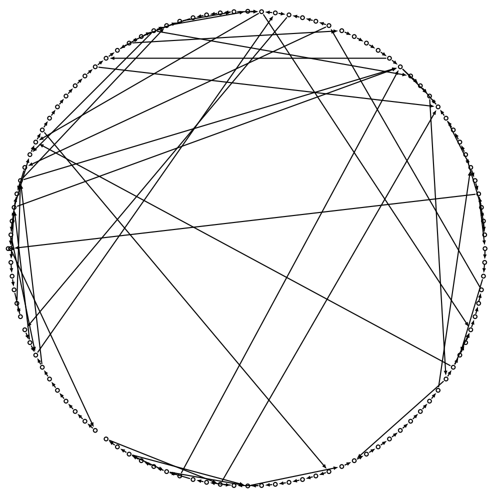
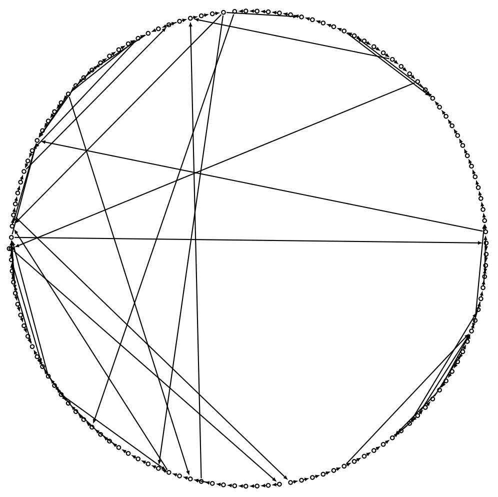
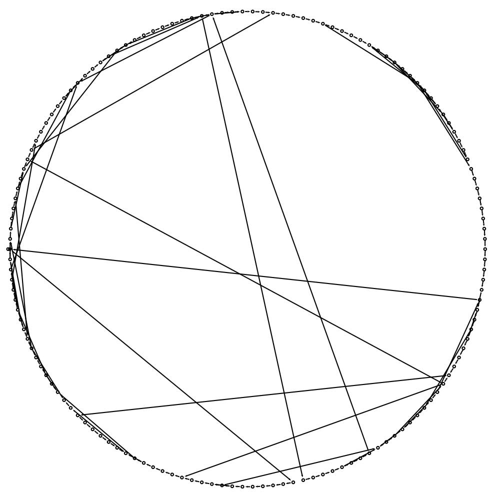

(d) $k = 4$

Figure A.1: Learned driver interaction models with $1 \le k \le 4$


(a) $k = 5$


(b) $k = 6$


(c) $k = 7$

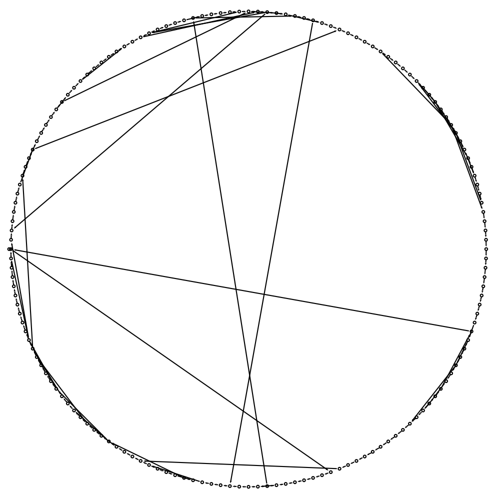
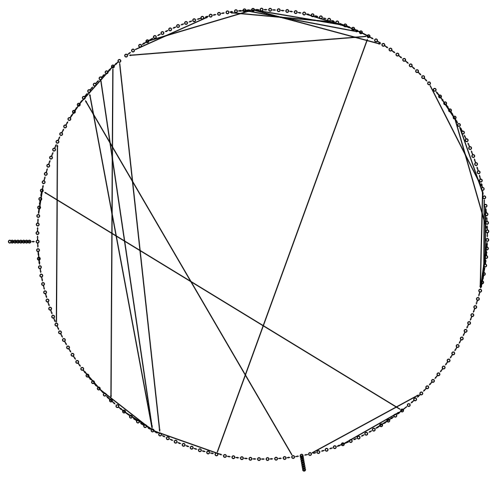

(d) $k = 8$

Figure A.2: Learned driver interaction models with $5 \leq k \leq 8$


(a) $k = 9$

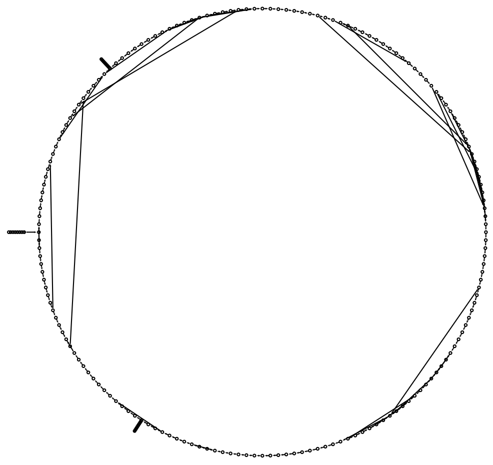

(b) $k = 10$

Figure A.3: Learned driver interaction models with $9 \leq k \leq 10$

List of Figures

List of Listings

List of Tables