Institute for
Software and Systems Engineering

Master's Thesis

Equivalence Analysis for Software Abstraction Layers

Henning Femmer

Software Engineering
Elite Graduate Program

Universität Augsburg University

LMU

TECHNISCHE UNIVERSITÄT MÜNCHEN

Institute for Software and Systems Engineering

Equivalence Analysis for Software Abstraction Layers

Author: Henning Femmer
Matrikelnummer: 1115407

First Supervisor: Prof. Dr. Wolfgang Reif
Second Supervisor: Prof. Dr. Alexander Knapp
Advisors: Dr. Dharmalingam Ganesan, Dr. Dominik Haneberg, Dr. Mikael Lindvall

Submission Date: March 16, 2012

Software Engineering
Elite Graduate Program

I assure the single handed composition of this master's thesis only supported by declared resources.

Munich, March 16, 2012

. . . .

(Henning Femmer)

Acknowledgments

It is a pleasure for me to thank those who made this thesis possible. First of all, I would like to show my gratitude for the great opportunity to work at Fraunhofer CESE (Center for Experimental Software Engineering) in Maryland. I am indebted to: Dr. Mikael Lindvall, for the various interesting discussions during and after work, which truly broadened my horizon; Dr. Dharmalingam Ganesan, for sharing his ideas and discussing mine; Christoph Schulze, for being able to learn with him during the experiments with KLEE, and numerous others who took their time to ask and answer. I would also like to acknowledge the NASA's SARP (Software Assurance Research Program) with Lisa P. Montgomery and her SARP team for partly supporting this work under the "Architectural Analysis of Dynamically Reconfigurable Systems" project.

Nor would this thesis have been possible without the support from my German advisors: Professor Dr. Wolfgang Reif, for establishing the contact to CESE; Dr. Dominik Haneberg, for making the exchange possible and for his support especially through the latter part of this thesis and Professor Dr. Alexander Knapp, to whom I owe my deepest gratitude for guiding and supporting me through good and bad times.

I cannot express in words how thankful I am to those who helped when help was needed: My parents Petra and Bernd, my sister Almuth, my brothers Bernd-Martin and Nils, as well as Virginia, Rob, Anto and Ben. This is to you.

Zusammenfassung

Die Herstellung von wiederverwendbaren Softwareframeworks, welche nicht an bestimmte Komponenten gebunden sein sollen, erfordert ein spezielles Design um jene Komponenten austauschen zu können. Diese Komponenten, die in Form von Betriebssysteme, Middlewares, Datenbanken oder Hardware Komponenten auftreten können, nennen wir den abstrahierten Aspekt.

Eine Weise diese Austauschbarkeit herzustellen, ist die konkreten Komponenten zu meiden und stattdessen die indirekte Nutzung via definierter Schnittstellen durchzusetzen. Zusammen mit den konkreten Implementierungen bilden diese Schnittstellen die Softwareabstraktionsschichten, die sicherstellen, dass die gleiche Funktionalität angeboten wird, unabhängig von der konkreten Konfiguration des abstrahierten Aspekts im Zielsystem, etwa der Hardwarekonfiguration oder des Betriebssystems. Da die Entwickler Software gegen das Interface anstelle der konkreten Implementierung entwickeln, nehmen sie an, dass ihre Software sich in jedem Szenario, unabhängig von der konkreten Konfiguration, gleich verhält.

Daher muss getestet werden, ob alle Implementierungen der Schnittstelle äquivalent sind um Differenzen festzustellen und zu entfernen. Aus dem Satz von Rice folgt allerdings, dass diese Äquivalenz nicht entscheidbar ist. Deswegen ist es nicht möglich ein Programm zu schreiben, dass für zwei beliebige Funktionen bestimmen kann, ob diese äquivalent sind. Dementsprechend müssen Heuristiken angewendet werden, die eine Näherungslösung für das Problem erstellen.

Diese Arbeit beschreibt und analysiert das oben beschriebene Problem im Detail. Es fasst verwandte Arbeiten zusammen und erzeugt eine Taxonomie möglicher Ansätze. Ein neuer Ansatz wird entwickelt, welcher auf Reviewing basiert und durch statische Analyse und Machine Learning unterstützt wird. Diese Methode wird anhand einer Fallstudie mit dem Operating System Abstraction Layer (OSAL) des NASA Core Flight System evaluiert, welche bereits in verschiedenen Missionen der NASA Anwendung findet. Der vorgestellte Ansatz entdeckte 111 Problemstellen, von denen sich viele als Fehler verschiedenster Schwere herausstellten. Einige der Problemstellen wurden bereits vom OSAL Team als Fehler bestätigt. Dabei wurden verschiedene Klassen von Problemstellen klassifiziert; eine Implementierungen etwa wiesen Inkonsistenzen im Rückgabeverhalten auf. Ein weiteres Beispiel zeigt sich in der unterschiedlichen Interpretation der erlaubten Länge der Namen gewisser Komponenten.

Die Ergebnisse deuten an, dass Machine Learning und Statische Analyse helfen können um Äquivalenz festzustellen. Weiterhin zeigt sich, dass ein Reviewing-Werkzeug, welches von Statischer Analyse und Machine Learning unterstützt wird, erfolgreich schwerwiegende Probleme in realen, sicherheitskritischen Anwendungen aufdecken kann.

Abstract

Creating reusable software frameworks where the goal is not to be bound to certain components often involves designing for exchangeability between common types of components, such as operating systems, middlewares, databases, and hardware components. We call this the abstracted aspect.

One way to enable exchangeability is to avoid using such components directly and instead promote indirect use through common interfaces. Together with their implementation, these interfaces form software abstraction layers, which ensure that the same functionality is delivered to the user, regardless of the concrete configuration of the abstracted aspect, for example the concrete operating system or hardware setup. As developers create software following the abstraction layer's interface instead of the concrete implementation, they assume that their software will always behave in the same way independently of the actual configuration in use.

Thus, there is a need to check whether all implementations of the interface are indeed equivalent, so that differences can be detected and removed. However, as a consequence to Rice's theorem, equivalence of certain software components is not decidable. Hence, no computer program can accurately determine whether two arbitrary functions are equivalent. Consequently, heuristics have to be created to target the problem.

This work first describes and analyzes the aforementioned problem in detail. It furthermore sums up related work and creates a taxonomy of possible approaches. A novel approach is designed, which is based reviews and supported through static analysis and machine learning. This method is then evaluated in a case study analyzing the Operating System Abstraction Layer (OSAL) of NASA's Core Flight System, which is commonly used in various space missions. The developed approach detected 111 issues of which many turned out to be bugs of differing severity. Some issues were already acknowledged by the OSAL team. We identified various classes of issues, for example, the implementations showed inconsistent return behavior at various points. To name a second, some functions interpreted the allowed length for names of certain components differently.

The results indicate that machine learning and static analysis can provide helpful support for determining component equivalence. Furthermore, the results show that a tool that supports reviewing by adding information from static analysis and machine learning, can be successfully applied to find serious issues in real world safety critical applications.

Contents

I Introduction 1

1 Overview over Equivalence Analysis for Software Abstraction Layers 3
1.1 Developing Flight Software at NASA 3
1.2 Arguments for Addressing Equivalence Analysis 4
1.3 Scope of This Thesis 5
1.4 Structure of This Work 5

2 Related Work 7
2.1 Existing Work on Software Equivalence 7
2.2 Discussion 9

II Analyzing Equivalence 11

3 Equivalence of Software Abstraction Layers 13
3.1 Software Abstraction Layers 13
3.2 Equivalence 17
3.3 SALs and Equivalence in the Scope of This Work 22

4 Analyzing Equivalence of SALs 25
4.1 Dynamic Code Analysis 26
4.2 Forward Symbolic Execution 28
4.3 Static Code Analysis 38

III Tool Support for Reviewing Software Equivalence 41

5 Detecting Differences in Code 43
5.1 Process of Detecting Differences 43
5.2 The Data Extractors 45
5.3 Summary 51

6 Classification and Filtering of Hints 53
6.1 Terminology 53
6.2 Classification of Function Pairs 54
6.3 Used Features 54
6.4 Used Classification Algorithm 55
6.5 Feature Selection for Filtering Hints 57
6.6 Overall Process 58

7 Using the User for Equivalence Analysis 61
7.1 Semi-automated Equivalence Analysis 61
7.2 Support in a Reviewing System 61
7.3 Complete Tool Process 63

IV Evaluation 65

8 Evaluation with the Operating System Abstraction Layer 67
8.1 Introducing OSAL 68
8.2 Is the OSAL a Representative Industry Project? 70
8.3 Is Traditional Diffing Appropriate to Find Equivalence? 71
8.4 Is the Presented Approach Able to Find Serious Issues in OSAL? 74
8.5 Is the Automated Approach Able to Classify Equivalence? 82
8.6 Threats to Validity of the OSAL Case Study 88
8.7 Summary 89

9 Conclusion and Future Work 91
9.1 Wrap-Up 91
9.2 Outlook 92

Bibliography 93

Appendix 101

A Mail Conversation with Alan Cudmore 103

B Example for Functional Delta 105

C Configurations 107

D KLEE Examples 111

E OSAL 119

List of Acronyms

API	application programming interface
BSP	board support package
CFG	control flow graph
CFS	Core Flight System
CSV	comma-separated values
HAL	Hardware Abstraction Layer
LRO	Lunar Reconnaissance Orbiter
LSP	Liskov Substitution Principle
OCL	Object Constraint Language
OS	operating system
OSAL	Operating System Abstraction Layer
POSIX	Portable Operating System Interface
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	real-time operating system
SAL	software abstraction layer
SDO	Solar Dynamics Observatory
SOA	Service Oriented Architecture
XML	Extensible Markup Language

Part I

Introduction

Chapter 1

Overview over Equivalence Analysis for Software Abstraction Layers

In order to introduce equivalence analysis for software abstraction layers (SALs), we describe a scenario of application, explain the origin of the problem and describe why the problem should be addressed. We furthermore define scope and structure of this work.

1.1 Developing Flight Software at NASA

In this section we describe how space missions develop software at NASA based on the reusable Core Flight System (CFS) code base as well as reasons for the need to analyze the equivalence of software abstraction layers.

Scenario

The basic requirements for flight software are very similar from mission to mission. Traditionally, missions have been ad hoc reusing flight software from older missions with similar hardware. The software was then adapted to the new mission, requiring extensive manual coordination and leading to very little reuse across NASA centers or external entities [GLA+09]. In order to create systematic reuse in NASA space missions, the organization has created various frameworks that incorporate standard mission components. One of these frameworks is the Core Flight System, which provides a flexible plug-and-play software architecture, including reusable libraries for flight software and an integrated tool suite [Bar09].

One advantage of using the CFS framework relates to the development process of flight software. In missions that base on CFS (such as Solar Dynamics Observatory (SDO), Lunar

Reconnaissance Orbiter (LRO) etc., see Section 8.1) developers can program on their Linux or Mac machine to create the needed software and run the necessary tests. Afterwards, the same code base can be deployed onto the real hardware, running under a real-time operating system serving the safety needs of space software.

This is achieved through the Operating System Abstraction Layer (OSAL), a small framework that is placed between the software and the operating system (OS). The OSAL contains an interface including most relevant system calls and implementations for four major operating systems (Mac OS X, Linux, Real-Time Executive for Multiprocessor Systems (RTEMS) and VxWorks6). Developers write software against the OSAL interface, enabling the system to be deployed to either one of the OSes without changing any code.

Problems in Software Abstraction Layers

Separating development from deployment assumes that the choice of platform makes no difference. For example, tests that are executed within laboratory settings should have equal outcome in the real setting to maintain their usefulness. Otherwise, the software, which is a supposedly deterministic component, creates unexpected behavior.

From a developer's point of view, he[1] will develop software based on the information provided by the system's specification. For example, after a file is created the system might want to check if the operation was performed successfully and thus needs OSAL's response. But what if this behavior, the information from OSAL, varies from operating system to operating system? The developer would need to now the underlying operating system, and would have to develop specifically for this OS. At this point we lose the operating system independence, which undermines the purpose of the whole OS abstraction layer. Instead, what we are looking for is consistency across the different implementations of the interface, so that any difference in the underlying OS remains unexposed to the user of the interface.

1.2 Arguments for Addressing Equivalence Analysis

Thus, there is a need for equivalent components, as well as methods that allow us to analyze whether two components are equivalent or not. However, the question is: Is it economically reasonable to create an approach that addresses equivalence analysis? The following three arguments support the hypothesis that this is the case.

First, by establishing a lightweight approach we can keep costs low. If a tool can be created that is easy to apply and finds a reasonable amount of differences between implementations (often a sign of bugs), the investment for this application is quite low compared to more labor intense strategies. We created an approach based on a combination of static analysis, machine learning and reviewing, where no additional set up (such as formal interface specifications or similar) is required to analyze the code base. Hence, the tool we developed demonstrates that costs for applying a lightweight approach can be very low. Additionally, first analysis results can be retrieved early after first investment.

Second, in safety critical systems every bug can turn into a serious problem threatening the life of human beings, e.g. in the Therac-25 radiation therapy machine that led to

[1] For pure readability reasons we will stick to the generic masculine during this whole work.

the deaths of five people in the 1980s. And even in unmanned spacecraft, when human beings are not directly endangered, bugs can threaten whole missions, such as the US$1 billion loss of the Ariane 5 in 1996. The software was tested using simulation software that was designed for Ariane 4, assuming the same results would hold for the successor. Hence, the real-time behavior in deployment was not equivalent to the behavior in the simulation, which prevented the developers from detecting the issue during development [LLF+96]. The use case analyzed in this thesis is functioning as the base for such safety critical systems.

Third, we knew about the demand: We discussed the issue with developers at NASA. They assured us that to their knowledge there is no such tool in use, but they would be very interested to analyze their software with respect to component equivalence for the above-mentioned reasons.

1.3 Scope of This Thesis

In this thesis we focus on equivalence of SALs where different implementations of an interface exist. An approach was developed that detects equivalence and violations thereof, in other words, a tool that determines whether implementations can be exchanged without any unexpected behavior. One could require the implementations to prove compliance to a given formal specification of the interface. We, however, developed an approach that does not require a formal specification of the interface, as this specification might not always be available. Instead, implementations are compared against each other.

The approach was especially developed to support real world scenarios where the system under analysis interacts with and depends on the environment in which the system is executed. This comes with two challenges: First, not all code of this environment is necessarily accessible, for example, because the system is running under a specialized, proprietary operating system. Accordingly, the developed approach needed to be able to analyze only parts of the system. Second, in real world scenarios systems also have a state, which stores certain information about the current condition of the system. Hence, the approach developed in this thesis can handle stateful systems, which initialize, modify and read from a global state.

1.4 Structure of This Work

This thesis consists of five parts (see Figure 1.1). Part I describes the frame of this work. It introduces into the topic, motivates the invested effort and defines the scope of the thesis. Furthermore, we give a little tour through existing work on the topic. Part II describes software equivalence analysis in detail and classifies possible approaches, of which one approach (symbolic execution) is explored. Afterwards, a second approach is elaborated in Part III in more detail. In Part IV a case study using the latter approach is conducted and results of the experiment are explained. We give a summary of the work presented in this thesis subsequently. Lastly, an appendix serves for references.

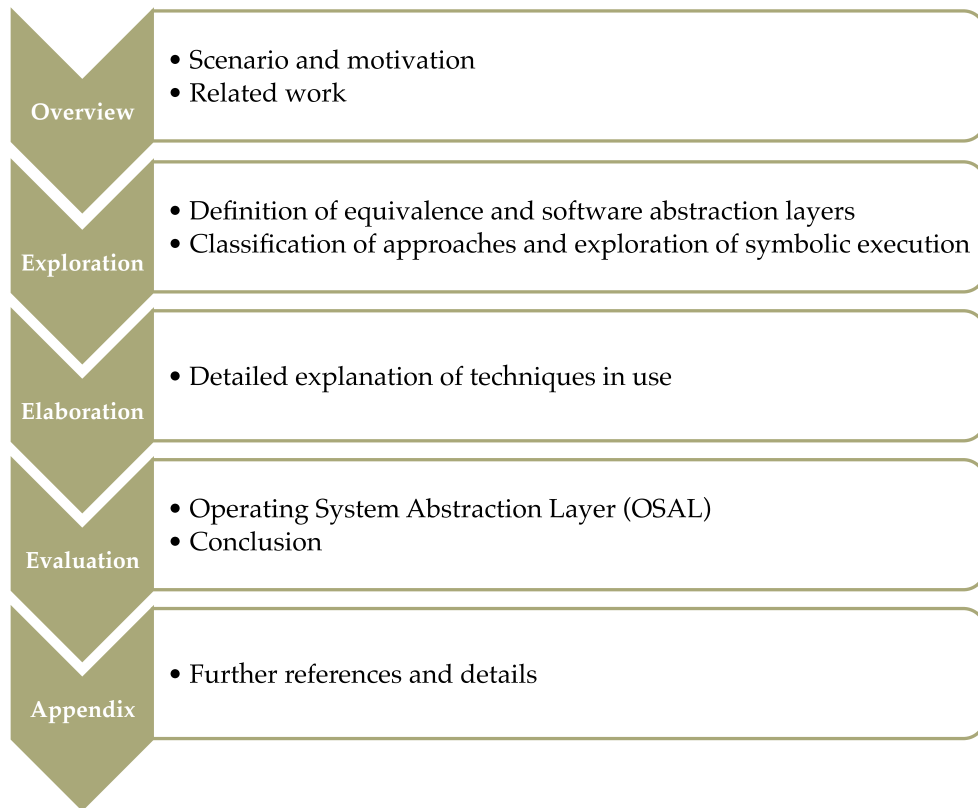

Figure 1.1: Structure of this work

Figure 1.2: Word cloud of this work

Chapter

2 Related Work

Relevant work from both industry as well as research is summed up. Furthermore we describe how the related work influenced the present thesis.

2.1 Existing Work on Software Equivalence

Determining whether two programs are equivalent is a problem reaching back into the 1950ies. The earliest resource we found was by Hoare [Hoa69], who references papers supposedly discussing the topic in 1958. Yet, for most of the time it stayed a theoretical problem. Recently, equivalence of components came back into focus with applications in hardware verification and regression analysis.

Equivalence in Hardware and Embedded Software Design

In hardware, developers often have to check if a developed model is consistent with a certain piece of hardware. This is a crucial task after optimizations and transformation of models into circuits. Accordingly, equivalence in hardware verification, namely equivalence of circuit implementations, finds wide application [LVH10] in its field. Due to enabling technology like Boolean Satisfiability Solvers and Binary Decision Diagrams, commercial tools are in common use (e.g. [Syn12]). Recently, there have been attempts to leave the circuit world and try to find equivalence in embedded software. For one, [FH05] compares equivalence of code in the field of embedded software. However, these approaches often have very strong requirements: For example, [AEF+05] split up the functions for finding so-called *cut points*, where verification is (computationally spoken) easier. From proofing cut point by cut point, they try to prove equivalence for the whole program. However, they assume that the source code contains neither recursion nor loops. Alternatively, in

bounded model checking and related approaches, loops are unrolled n times, in order to escape the undecidability problems arising from unbounded loops (e.g. in [CKY03]).

Equivalence of Web Services

In a different field, web services serve as highly decoupled components for providing data or serving for certain tasks. With the goal of enabling comparison of Web services, [ILF09] presents a middleware for service integration. It includes a component, which is checking the equivalence of non-functional requirements (the Quality-of-Service, QoS) of the services included in the framework. One could imagine checking non-functional requirements of different implementations similarly to non-functional requirements of Web services. We will explain this in Section 3.2.1.

Program Element Matching and Semantic Differencing

[KN06] is a summary of the state of the art in the field of program element matching. The paper evaluates the different methods available based on hypothetical change scenarios. However, their goal is to support the development of software that origins in cloned parts, especially focusing on the evolution of information hiding. Their work is supposed to help determining whether a bug has an impact on more than one piece of code. [BCJ07] suggest dependence highgraphs, a hierarchical representation of the source codes control and data flow, as an intermediate structure in compilers and source code analysis. They claim that program element matching is easier to perform with this representation.

Using textual diffing for regression analysis is presented in [Hor89]. Furthermore, enhanced control flow graphs (CFGs) are used to identify semantic changes. Other early work on semantic diffing is described in [LS92]. The authors define an algorithm that identifies changes in CFG and reduces the graph to the relevant part afterwards. This work is transferred to Java and extended with object-oriented semantics by [AOH06].

Another more recent group focusing on differences between code versions is the SymDiff (*Symbolic Differencing*) project at Microsoft. The goal is to extract clear information displaying the semantic differences between versions. This is useful to prevent introducing undesired side effects. In [KLR10] they define *conditional equivalence* as two functions being semantically equivalent under certain inputs and implement a basic version for the intermediate language BoogiePL. The Z3-SMT Solver is used for verifications. In [HKLR11] the authors try to automatically extract termination conditions and summaries from the source code. They demonstrate the ideas on a handful of laboratory examples.

Equivalence Using Symbolic Execution

[Kin76] introduced forward symbolic execution. We do not intend to list the general state of the art of this field here, it can be found in [CKP$^+$11] or [SAB10]. The common tools used for symbolic execution are *JPF-SE* [APV07] for Java, *s2e* [CKC12] for symbolic execution within the real environment, and *KLEE* for C code [CDE08]. The latter also includes a short example of comparing two functions for functional equivalence. This example is extended in [RE11], analyzing more libraries for differences. In their work the authors compare two functions by finding, storing and comparing the transformations of a program. However, most of these applications cannot handle common real life code

elements such as structures or external routines depending on symbolic arguments (such as system calls).

Differential symbolic execution [PDEP08] formalizes the functional equivalence and analyzes different versions of one program. It extracts only differing parts and performs symbolic execution on these, thus reducing the state explosion problem. Similar approaches are presented in [MSF05] and [SMS+07].

Equivalence for Differential Static Analysis

[LVH10] sums up the state of the art in differential static analysis. This field analyzes different versions of one program at compile time. The paper describes the challenges and advantages of using static analysis and names applications for differential static analysis.

One group at Technicon in Haifa, Israel worked on verifying equivalence of two versions of a program (*Regression Verification*). One mentioned use case is refactoring, where proofing functional equivalence could serve as a safety net. [GS08] introduces formal rules for equivalence. They define six different definitions of (functional) equivalence and show how to prove them for an artificial language LPL. These are further implemented in the Regression Verification Tool (*RVT*) described in [GS09]. Bounded model checking is used to verify equivalence.

Equivalence Through Dynamic Code Analysis

In contrary to all papers mentioned before, [HEJ09] apply dynamic code analysis. The authors show how to compare execution traces of different program versions. They make use of aspect-oriented programming and analyze Java programs that use a certain subset of the Java language. Applying dynamic testing for functional equivalence has been proposed by some authors (e.g. [Vou90]). They suggest a technique called back-to-back testing, where behavior of functional equivalent versions of a software system during testing is compared. They work with a probabilistic model of occurring events during testing to determine the functional equivalence. In this they apply a certain tolerance. However, it is important to note that this tolerance does not compare with our terminology of abstracted aspects: The tolerance provides a way to state that two systems are 90% equivalent with no regards of where the difference exists. In contrary, the abstracted aspect requires the software to be 100% equivalent on domains outside the abstracted aspect and not equivalent within the domain.

2.2 Discussion

Most of the work described in this chapter addresses a problem that is very similar to the one we were facing with equivalence for software abstraction layers. Some were very close in the goals. But none was handling the real world issues we were focusing on: None of the above-mentioned papers deals with the tolerance we need in our approach. As software in our case accesses subcomponents that are potentially not accessible during analysis, we need to work with a broader term of equivalence than the ones given in related work (for example in [PDEP08], see Section 3.2). Also, software running on different systems (such as different hardware, operating systems, parsed by different compilers and so forth)

is analyzed, a special case we could not find anywhere in related work. Also, many of the more formal approaches do not present any results from real industry projects: The examples presented usually do not include any side effects, the initialization of a global state or function calls to external libraries. Our experiences with symbolic execution support the hypothesis that these very exact approaches are not the perfect choice in a setting with much uncertainty. We describe the challenges, findings and results in Section 4.2. However, preliminary findings were not very promising; consequently, we chose to use heuristics and static analysis to address the problem instead.

We furthermore filter the results of the static analysis approach. This is an idea that has been successfully applied before (e.g. in [BLH08]). However, to all our knowledge no research has been conducted to apply data mining or machine learning for functional equivalence of software systems.

The approaches applying dynamic analysis such as testing or trace analysis could not be applied in our case study, as execution of software was not possible. Hence, we focused on static analysis and forward symbolic execution. This decision is explained in more detail in Section 4.1.

The works on symbolic differencing inspired the tool developed. The structure of our analysis framework is very similar to the one described in [NNHME09], although not as comprehensive. The concept of beliefs, introduced in [ECH+01] is conceptually similar to our *hints*, explained in Section 5.1 in the way that it provides evidence but does not guarantee a fact. We gave the concept a different name, as they differ structurally and in the way they are extracted.

Furthermore, we built our definitions of equivalence for software abstraction layers atop of the definitions provided in [Rom02], [ILF09], [LVH10] and finally the more formal definition in [PDEP08].

Part II

Analyzing Equivalence

Chapter 3

Equivalence of Software Abstraction Layers

We provide an overview over software abstraction layers (SALs). The reasons for implementing and the actual implementations of SALs are analyzed, their structure is elaborated in detail and an according equivalence definition is given. Afterwards, different aspects and definitions of equivalence are identified and explained. Lastly, we will point out which direction of the field we will explore in the rest of the thesis.

3.1 Software Abstraction Layers

Software abstraction layers are common parts of software architectures. They serve a well-defined purpose and can appear at various layers of software systems, which we will show in the following examples.

3.1.1 Purpose and Architecture of a Software Abstraction Layer

When developing software, various situations necessitate different implementations for a similar aspect: We want to switch between different search algorithms, a web mesh up page accesses different service providers or a developer creates a program that is supposed to run on different hardware or operating systems. In all of the above-mentioned cases we have a system and various subsystems providing information or services to the top layer. In all of the cases there is an implicit or explicit specification of the sublayer's requirements, either explicit through formalized interface definitions like contracts or OCL constraints, or more often implicit through naming, rough textual documentation or discussions between developers. And in all of the cases, the sublayers are similar besides one aspect. We call this the *abstracted aspect* or *variation*.

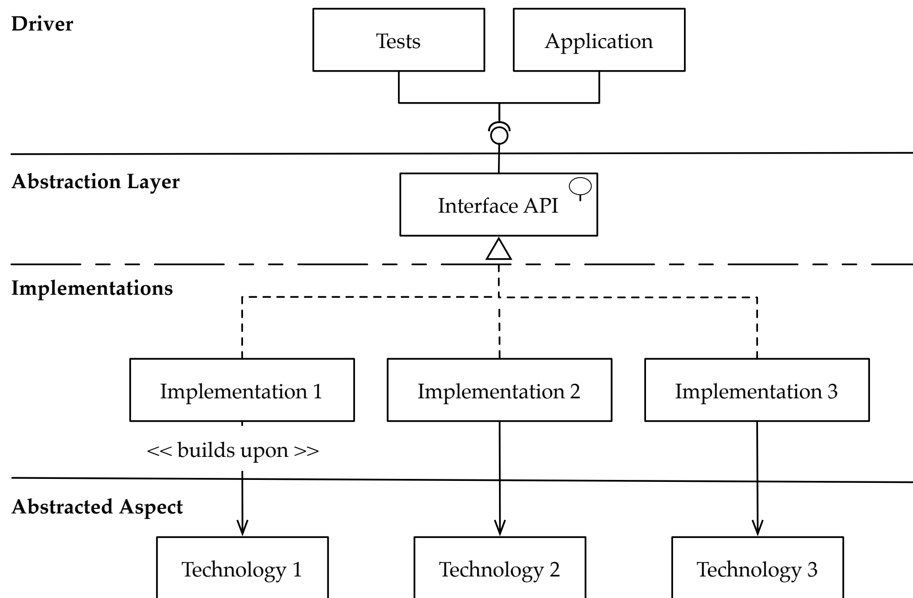

Figure 3.1: Architecture of abstraction layers

Abstraction Layers are needed when we want to include this possible variation in the architecture of our system. Architecturally spoken, we want to hide the concrete implementation of the aspect underneath and thus define an interface that is used in further development (see Figure 3.1). The interface defines the purpose of the implementations underneath. The implementation of this interface can be exchanged as long as each implementation follows the interface definition. We call the application that uses the interface *driver*, the interface represents the *abstraction layer*, and *implementations* that build upon a certain technology represent one possible choice in variation.

3.1.2 Aspects of Abstraction

Various abstraction layers are part of our day-to-day software. We demonstrate the widespread use of SALs in various applications by giving a handful of examples.

Hardware Abstraction

The first example is part of many major operating systems: It is called the Hardware Abstraction Layer or HAL. Large operating systems like Windows NT or distributions using the Linux kernel have a certain layer in the core that enables the OS to forget about specific hardware details (see Figure 3.2). For example, it does not matter which supplier produced a certain drive and it also does usually not matter which graphic card is located on the mainboard. Otherwise the developers of the OS would need to change the code of the operating system for every new piece of hardware that is supposed to be compatible with the system.

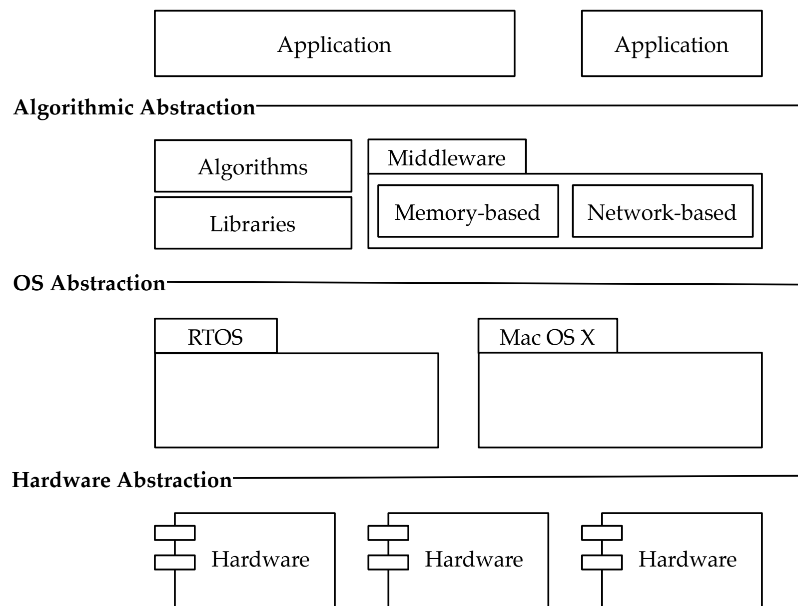

Figure 3.2: Different levels of abstraction layers

Operating System Abstraction

The second example lifts the idea by one layer and tries to find abstraction not only from hardware, but also abstraction from the operating system (for example real-time operating system (RTOS) and Mac OS X, see Figure 3.2). There are two approaches towards this challenge:

One approach defines an application programming interface (API) that includes every functionality an operating system needs to provide to an application. Programs develop and compile against this interface. This interface is implemented for each operating system and stored within the runtime system. The program then dynamically loads the library during runtime, and can thus access the system specific functionality, without changing a line in the code. The implementations of the interface are created by encapsulation of the operating system through wrappers around the system calls. This lightweight approach will be part of our case study and explained in detail in Section 8.1.

The second approach for abstracting from operating systems is building a virtual machine. This virtual machine is an operating system inside the operating system. All programs develop and compile against the virtual machine, which translates the program into virtual machine bytecode. The virtual machine bytecode is then translated into operating system specific machine code during runtime. The virtual machine approach is heavily in use within the java runtime environment, the standard for developing operating system independent code.

Both approaches unites that they standardize the interface between OS and applications. They differ in the way this interface is implemented - either lightweight through wrappers and libraries or by building a complete virtual machine with an intermediate language.

Remark: The Portable Operating System Interface (POSIX) standard family is an approach to create such an API. The operating systems share the same API, so that developers

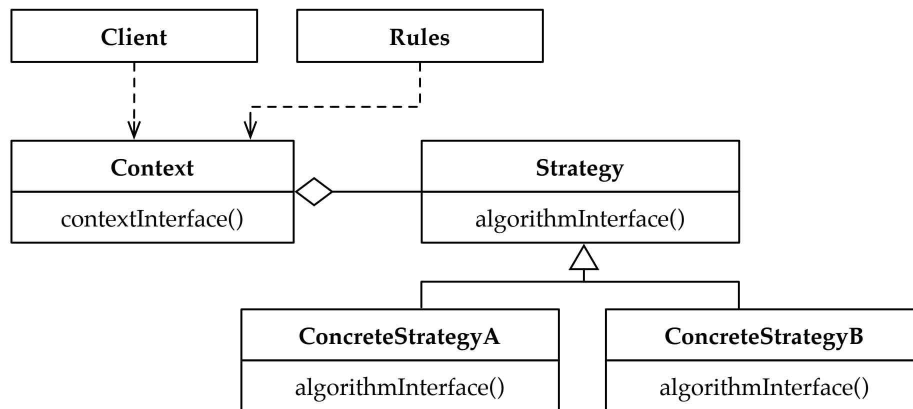

Figure 3.3: The strategy pattern

can access functionality through the POSIX interface. RTEMS and VxWorks6 are fully compliant, just as Mac OS X, Cygwin and Linux systems are largely compliant. However, the devil is in the details. For example, [Jos03] lists the small but relevant semantic differences between the POSIX standard and the Linux kernel implementing it. [Wal95] works out POSIX' problems and names a major issue: It is too small to provide the full access (e.g. to underlying hardware) and all necessary system specific components, such as a GUI. Upon request, Alan Cudmore, the initiator of the NASA OSAL (see Section 8.1) names three major reasons why POSIX is not a sufficient OS abstraction layer[1]: First, POSIX is sometimes implemented inconsistently. For example, in the concrete implementations the original OS system calls provide more consistent return codes. Second, the implementations of the original OSes are clearer and more intuitive than the POSIX standard, for example in the domain of semaphores. And third, not all functionality is implemented in all POSIX OSes. Cudmore names timed message queues as an example. Consequently, in order to provide fully consistent POSIX APIs one can either create workarounds for the known differences or set up a completely new, consistent layer that only uses the POSIX standard where appropriate (for the full conversation see Appendix A).

Algorithmic and Library Abstraction

We can continue lifting the abstraction up to a third level: abstraction within the application's source code (see Figure 3.2).

This is the case if we want to use different algorithms. Exchange of algorithms often manifests itself in the architecture with the Strategy pattern, with the intent given by Gamma:

> Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. —[Gam95, page 315]

Strategy patterns are used to decouple certain task from the performing class. This enables reuse and exchange of algorithms. However, when a strategy pattern is used to

[1]Personal communication

exchange the algorithm, equivalence analysis could help to prevent unexpected behavior. For example, assume two different implementations exist for a query: One implementation is designed for SQL conform databases, the second implementation retrieves the same information from a NoSQL database. These implementations will cause differing actions on the database, as the databases follow completely different paradigms. However, the result of the implementations should be equivalent. For instance, if for any reason the database query cannot be executed, all implementation should respond with an equivalent behavior, so that the driver (i.e. the calling function) is really decoupled from the concrete implementation. If the implementations were not equivalent, the calling function would need concrete references to the implementations, e.g. a branch in the calling code that includes the error codes of the NoSQL definition.

The intention to exchange the strategy can arise for different reasons: Above, we mentioned different technologies (i.e. database systems), another reason is the requirement to exchange libraries, e.g. because of runtime constraints, or even the need to exchange whole layers. For example, the system might need to switch between a shared memory and a network-based middleware system (as in [GLR+10]), depending whether the concrete implementation is deployed within the same machine or in a distributed setting. Both systems should have the same behavior, except the medium used for transmission of messages.

In summary, software abstraction layers are placed between various layers of software systems, from the very top to the very bottom. Various applications exist; yet, to all our knowledge, checking equivalence is not very common.

3.2 Equivalence

Implementations of abstraction layers are equivalent, but not equal. Differences might lie in the algorithm or library used, the web page, the operating system or the hardware. The software will do things differently; for example, it will send network packages to a different address or it will use different protocols for the different hardware in use. Hence, the implementations are not equal; they differ. However, the implementations have the same purpose and it does not change the overall system when we switch the implementations. Thus, no matter if I use hard drives or floppy disks, storing of data should have the same results. Hence, we say that the implementations are equivalent with a definition of equivalence given in the following sections.

In contrary, we refer to equal implementations when two implementations do exactly the same thing in exactly the same way.

> **Definition** Two functions are (syntactically) equal, iff the code of two the functions is characterwise equal.

We will discuss the frequency of equal functions in Section 8.3. However, the set of equal functions is a subset of equivalent functions.

$$\forall f, g : f =_{code} g \Rightarrow f \equiv g$$

In words: Every function pair of equal functions is also equivalent.

In order to define the term equivalence more precisely we looked into the field of requirements engineering. Here requirements are classified into two groups: Functional and non-functional requirements. Whereas non-functional requirements "describe aspects of the system that are not directly related to the functional behavior of the system" [BD07, page 160], functional requirements "describe the interactions between the system and its environment" [BD07, page 159].

We propose the same categories for defining equivalence.

3.2.1 Non-Functional Equivalence

Non-functional differences are very common when comparing SALs. Usually, they are built into technology and architecture or part of the aspect of abstraction. For example, later on we will analyze an operating system abstraction layer where we compare one implementation, which uses the commercial real-time operating system VxWorks6, with another implementation that uses the open source system RTEMS. It is common sense that these systems differ in their non-functional requirements, for example in their usability or reliability. Hence, non-functional differences are built into our code by choice of technology already. In the above-mentioned case minor differences are acceptable, depending on the space mission that uses the corresponding operating system. However, it might make sense to define the range of difference that is allowed. For example, it would be a good idea to find a metric for reliability and limit the non-functional differences in these regards.

Model-driven Development

In the field of model-driven development, Romberg defines equivalence of non-functional requirements for checking code against models:

> **Definition** Non-functional equivalence is the correspondence of estimates for non-functional values (such as timing properties) drawn from the model with actual properties of the implementation. [Rom02]

If we understand one piece of code as the model, this definition gives a rough idea of what we are looking for.

Service Oriented Architectures

A related definition in the field of Service Oriented Architectures (SOAs) is a little more formal. In SOAs, quality of service is a common term, defining non-functional requirements of services. [ILF09] created a meta-measure for calculating similarity of non-functional properties for services:

$$\text{QoSDegree}(op_m, op_n) = \sum_{i=1}^{|NP_{op_m}|} w_i * deg(np_i(op_m), np_i(op_n))$$

where QoSDegree is the non-functional equivalence degree between two functionally equivalent operations op_m and op_n, and w_i are weights for each QoS property np with index i of the operation op_m. NP_{op_m} are all non-functional properties of the operation op_m.

deg is some function giving a similarity level of two function's properties (e.g. arithmetic difference for real value properties).

This definition could be translated to software abstraction layers. Even though this formula is very vague, we think it might serve as a very useful framework for future work. The most interesting questions are yet to be answered though. Which quality measures should be used, how do we compare these measures, which weights should we give and what could be a reasonable degree of similarity permitted? We figure that the field of software metrics can answer at least some of the questions.

Conclusion

For systems with hard requirements, as it is often the case with RTOSes, the analysis of non-functional requirements is often compulsory. In RTOSes the developers often depend on explicit timing constraints to be met. Research on systematic non-functional equivalence analysis in cases where such an explicit specification does not exist, might lead to an interesting analysis to carry out before exchanging system components. We do not know of any research in this field. However, due to the limited scope of this thesis we focus on the area of functional equivalence.

3.2.2 Functional Equivalence

We will first present two definitions from related work, which will be translated into our domain in the second section. However, these definitions cannot lead to a perfect algorithm directly, because functional equivalence is undecidable, which we will explain in the third section.

Defining Functional Equivalence

To show different approaches we would like to present two definitions for functional equivalence. They differ in their degree of formality. The straight-forward definition gives a good gut feeling for the problem:

> **Definition** Functional equivalence relates methods that have the same externally observable behavior. [PDEP08]

The functions[2] should react equivalently under equal inputs, which includes the general system state if it has an impact on the function. The system state includes referenced memory location, file system state and all other external information that influences the function in its behavior. Furthermore, the externally observable behavior of two functions includes the return value as well as manipulations of the global state, such as variables or output streams.

[PDEP08] also give a more formal definition for functional equivalence based on symbolic summaries. Their work relates to symbolic execution and uses so-called symbolic variables, which are variables that do not contain concrete values, but logical expressions from the manipulation in the code (see Section 4.2).

[2]We use the term functions instead of methods, as we are mostly refering to non-object-oriented C code.

Partition	Effect
path == NULL	RETURN == INVALID_POINTER
strlen(path) ≥ MAX_NAME	RETURN == PATH_TOO_LONG
strlen(path) < MAX_NAME	RETURN == SUCCESS

Table 3.1: Example for a symbolic summary of a function with one string parameter

Definition A *partition-effects pair* (in, ef) consists of: an input constraint in, which is a conjunction of relational expressions defined over constants and symbolic variables, and an effects constraint ef, which is a conjunction of expressions that equate written locations to expressions defined over constants and symbolic variables.

Partition-effect pairs can be understood as summaries of the input and output relation of the function. Each line in the example in Table 3.1 is one partition-effect pair[3]. The input contains both parameters and the global state. The effect expressions contain expression about both the return value and the side effects of a piece of code.

These pairs can then be summarized to represent a function. Table 3.1 is an example for such a summary. However, to prevent contradictions and indeterminism, we have to take care that the input partitions are disjoint. This way for every input there is at most one pair (in, ef) in the summary:

Definition A *symbolic summary*, for a method m, is a set of partition-effects pairs $m_{sum} = (\text{in}_1, \text{ef}_1), (\text{in}_2, \text{ef}_2), \ldots, (\text{in}_n, \text{ef}_n)$ where the input constraints are disjoint, i.e., $\forall i \in [1, n], j \in [1, n] : i \neq j \rightarrow \text{in}_i \land \text{in}_j$ is unsatisfiable.

If the symbolic summary does not contain any contradictions, we can translate it into mathematical logic. This way we can use a precise logical calculus to talk about equivalence.

Definition Given a method, m, and its symbolic summary, m_{sum}, the *logical method summary* for m is

$$\langle m \rangle = \bigvee_{(\text{in},\text{ef}) \in m_{sum}} \text{in} \land \text{ef}$$

With this term we can now define functional equivalence as the logical equivalence of the logical method summaries.

Definition Given two methods m and n, m is *functionally equivalent* to n iff

$$\langle m \rangle \Leftrightarrow \langle n \rangle$$

Whenever two functions are not functionally equivalent we can compute the difference, the so-called functional delta:

[3]Please note that we added the function strlen(char*) for simplification. Functions are not part of the standard definition of partition-effect pairs.

Definition Given two methods, m and n, the *functional delta* is

$$\Delta_{(m,n)} = \langle m \rangle \wedge \neg \langle n \rangle$$

You can find an example for determining the difference of two functions with the functional delta in logics in Appendix B.

Functional Equivalence for SALs

However, even though very precise, these definitions do not include the abstracted aspect. But as mentioned earlier, in SALs it makes a difference where differences are located in relation to the abstracted aspect.

> **Definition** Two functions m and n are *functionally equivalent modulo an abstraction aspect A* iff all semantic differences between m and n are side effects in the domain of A.

We add the term *in the domain of*, which depends on the abstracted aspect. For example, if the abstracted aspect is a database system, differences in side effects may only affect the database system.

Rice's Theorem and its Consequences

However, functional equivalence is a special case of Rice's theorem and hence is an undecidable problem (e.g. [BCJ07]). Rice's theorem states:

> Any non-trivial property about the language recognized by a Turing machine is undecidable. —[CDW04]

Trivial properties are those that hold for either all functions or no function. As equivalence is non-trivial (there is at least one example for equivalent functions and one example for non-equivalent functions), functional equivalence is undecidable.

Consequently, we have to over- and/or under-approximate the problem, meaning we will incorrectly guess equivalent function pairs to be different and different function pairs to be equal. Having Rice's theorem in mind, it might be more appealing to build a tool that finds violations of equivalence, instead of building a tool that proofs equivalence, for two reasons:

a) Finding differences is computationally easier, just as it is often easier to find a counter example than proving a theorem.

b) When a difference is found, we found a concrete spot for improvement of software. This is often close to bug finding, which is very valuable to the developers.

Accordingly, in Part III we will focus on heuristics for finding differences violating the definitions above.

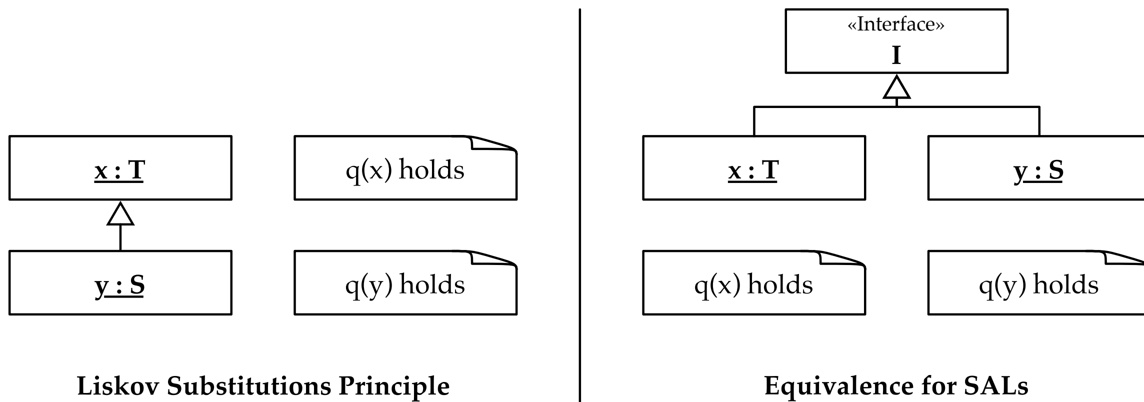

Figure 3.4: LSP vs. equivalence for SALs

3.3 SALs and Equivalence in the Scope of This Work

This chapter discussed Equivalence of SALs in a very abstract way. Due to time constraints this thesis can only focus on addressing a subproblem.

Domain

Whenever an interface is build in a way that the behaviors of sublayers are transparent but the result is defined, we want the implementations to be functionally equivalent. All implementations should be substitutable. This is very close to the Liskov Substitution Principle (LSP), which describes how subclassing should be performed (see Figure 3.4):

> **Definition** Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type S where S is a subtype of T. [LW94]

In other words: The external behavior of a subclass should be consistent with the behavior of its superclass. If the interface (which is the superclass in this terminology) contains a formal specification, we could consider to formally verify that the implementations are correct with respect to the specification. Yet many interfaces do not have a formal specification. Therefore, in this work we will focus on interfaces that are not explicitly formalized. Without formal specification we have to compare the different implementations of the interface against each other (see Figure 3.4, right hand side). Furthermore we want to reduce the LSP, as we are not interested in all properties, but only the externally visible ones. We could rewrite the LSP accordingly:

> **Definition** Let $q(x)$ be a property about objects of type T that are implementations of an interface I. Let $q(x)$ be visible to a certain external observer o that has no access to the domain of the abstracted aspect A. Then $q(y)$ should be visible to o for every object y of type S that is an implementation of interface I.

The exact definition of the observer o remains in the domain of the abstracted aspect.

Hence, the focus of this work is comparing implementations against each other where specification is only implicit or informal.

Granularity

Until now we mostly discussed equivalent implementations, thereby talking about the implementation at system level. However, APIs can be represented as a set of functions. Accordingly, one can brake up an API into its functions and discuss equivalence on the function level. This differs from the aforementioned view in the way that the function is looked at on its own, instead of analyzing all possible combinations of functions. Consequently, if there is a difference at function level, there is also a difference at system level. Also, in case we can determine equivalence (including all side effects) at function level, we know that the implementations are equivalent.

> **Definition** Two implementations of an API A, Im_1 and Im_2, are equivalent iff each function f defined in A has an implementation f_1 in Im_1 and an implementation f_2 in Im_2 and f_1 and f_2 are equivalent.

Consequently, we will focus on equivalence at functional level.

Summary

Software abstraction layers are APIs that hide certain abstraction aspects. Equivalence of SALs should be checked when no explicit specification of the interface is provided, but instead the implementations provide implicit specifications. Equivalence of SALs can be reduced to compare single functions for equivalence. It can furthermore be split up into two parts: functional and non-functional equivalence. Non-functional equivalence relates to non-functional requirements and could be compared using the appropriate metrics. However, the focus of this work is functional equivalence, which is defined as the lack of differences in externally visible behavior, such as return values and side effects. As determining functional equivalence is an undecidable problem, we suggest to focus on heuristics for determining the opposite of functional equivalence, i.e. functional differences.

Chapter 4

Analyzing Equivalence of SALs

Dynamic and static analysis "are typically seen as distinct and competing approaches with fundamentally different techniques and technical machinery" [Ern03]. Consequently, we categorize possible approaches into these categories (for an overview see Figure 4.1). We will first describe some ideas of applying dynamic code analysis. Second, we analyze our experiments with forward symbolic execution, a mixture between static and dynamic analysis. An introduction into forward symbolic execution and its application for equivalence analysis is given and preliminary results are explained. Lastly, application of static analysis is evaluated in terms of possible methods, advantages and disadvantages. The chapter finishes by describing tool-supported reviewing, the approach used throughout this thesis.

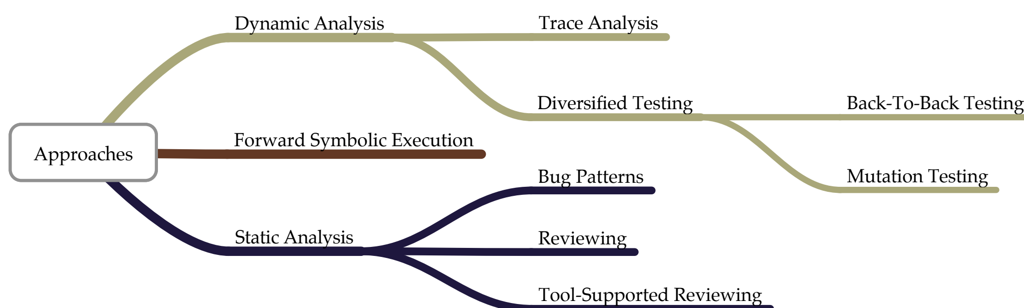

Figure 4.1: Presented approaches

```
2010-09-22 13:24:26.580 04916 FINE   logger.cpp             040 -> addRefCount logging already initialized
2010-09-22 13:24:26.580 04916 FINE   logger.cpp             040 <- addRefCount logging already initialized
2010-09-22 13:24:26.580 04916 INFO   sijabserver.cpp        072 -> WinMain -Embedding
2010-09-22 13:24:26.596 04916 FINE   sijabserver.cpp        033  -> CSIJABserverModule  RunMessageLoop
2010-09-22 13:24:26.596 04916 FINE   sijabserver.cpp        040      CSIJABserverModule  window message: message=1024, lParam=1492372, wParam=47806
2010-09-22 13:24:26.596 04916 INFO   javamanageratlobject 012   -> Loggable  CJavaManagerATLObject
2010-09-22 13:24:26.596 04916 TRACE  javamanageratlobject 014       Loggable  s_Instance == NULL: true
2010-09-22 13:24:26.596 04916 FINE   registry.cpp           012    -> CRegistry  CRegistry
2010-09-22 13:24:26.596 04916 TRACE  registry.cpp           017        CRegistry  RegOpenKeyEx(keyroot, path, 0, access, &m_key) == ERROR_SUCCESS: true
2010-09-22 13:24:26.596 04916 FINE   registry.cpp           012    <- CRegistry  CRegistry
```

Figure 4.2: A trace file

4.1 Dynamic Code Analysis

In dynamic code analysis, code is executed to be analyzed. We can analyze the runtime behavior of software either during the regular execution in the production environment, or alternatively we could run the software specifically for analysis as in tests. In any case, the software is run in a context equal or similar to the real production environment. This includes the whole deployment environment in both hard- and software.

Using Dynamic Analysis to Determine Software Equivalence

Two approaches for analyzing equivalence dynamically came to our mind: trace analysis and diversified testing.

Trace Analysis

To apply trace analysis we must observe runtime behavior during execution. Therefore, code traces are created that log what is happening in the program. Examples of such a log can be found in Figure 4.2. These logs describe the behavior in so-called events, which are combined in a trace. A set of traces describes the behavior of the program.

Very often aspect-oriented programming is used to add the logging of events into the original code. In aspect-oriented programming one can attach code (the so-called *weaving*) that is afterwards executed in specified situations, such as after or before function calls, assignment of variables etc.

In order to perform trace analysis, first of all the traces need to be created. In order to do so, we need to define events that create a log. With reference to the equivalence definition given in 3.2, we should focus on finding differences in externally visible behavior. As such, we would create events upon manipulation of external space, such as hard drives or sockets, and global variables, as well as function behavior such as function calls, input and output.

Second, we need to execute the software. Test cases might be useful, as they reflect various different applications of the software and hence create many different traces. Otherwise regular execution of the software should be logged.

Third, after the creation of traces, we could create state transition systems and check the different implementations for bisimilarity, which would focus on the behavior of the implementations. Alternatively, we could also compare the input/output/side effect triplets with tools that approximate the program's invariants, like DAIKON [EPG+07].

Diversified Testing

The second approach we could apply is (dynamic) testing. The field of diversified testing is especially interesting as it compares different versions of software. We present two suggestions for diversified testing: back-to-back testing and mutation testing.

Back-to-back testing [Vou90] is an approach with a goal very similar goal to ours, even though it originates in a different use case: It was designed to reduce discrepancies in multi-version redundancies as created for fault-tolerant software. The idea is simple (see Figure 4.3): In back-to-back testing, test cases are run on k functional equivalent versions of software. Afterwards differences in test results are investigated and corrected if necessary [Vou90]. Experiments have been conducted for $k = 2$ to $k = 13$ showing high potential to find discrepancies.

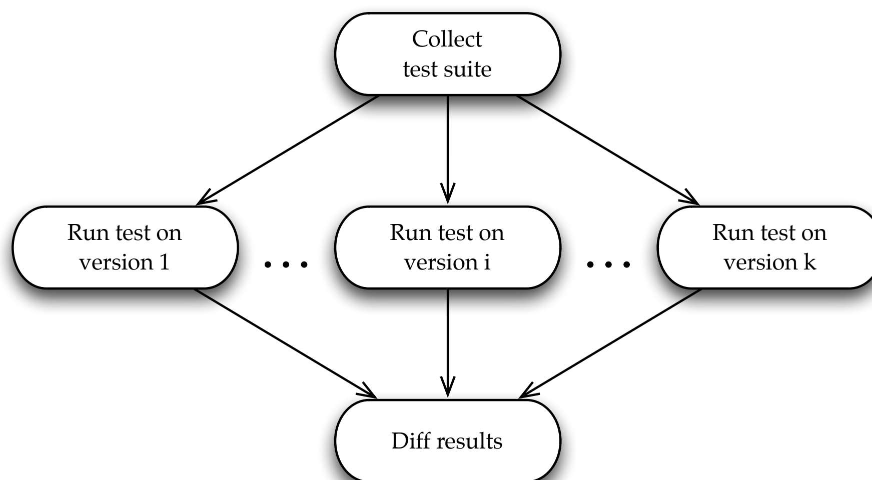

Figure 4.3: The idea of back-to-back testing

In mutation testing code is manipulated to check the sufficiency of test data sets. In order to do so, a test set is considered of appropriate size if mutation of code does always lead to a different result in the test suite, e.g. $\exists x \in T_i . P_0(x) \neq P_m(x)$ for a test suite T_i, an original source code P_0 and a mutation P_m. Mutation testing involves an easy process (see Figure 4.4) that involves the following steps:

1. Mutate the program under test P_0 into a mutated program P_m following a pre-defined pattern (the so-called mutant operator). For example, change variable references or replace == with `!=` conditions. Detailed replacements are discussed and can hence be found in related work, e.g. in [DR90].

2. Run the test suit. Determine if the test results differ between P_0 and P_m.

3. If the results differ, consider this mutation dead.

4. If the results do not differ, consider this mutation alive and extend the test suite. Return to step 2.

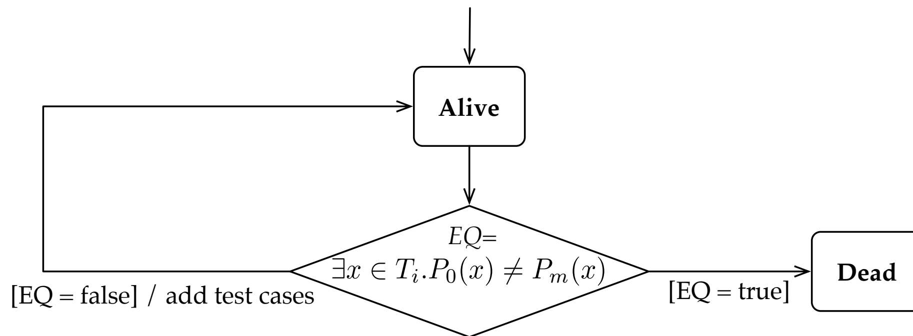

Figure 4.4: Lifecycle of a mutation

A combination of these approaches could be used for equivalence analysis for software abstraction layers. As we have *k* different functions, a comprehensive test set could be created by applying mutation testing onto both approaches and then afterwards checking both functions with the whole test set. Hence, function one is tested against the (implicit) specification of function two and vice versa. In a second step, differences in test results could be interpreted with the knowledge of back-to-back testing. However, we figure that probably an additional step needs to be inserted to deal with differences in test results resulting from the abstracted aspect (see Section 3.2.2).

Reasons for Not Applying Dynamic Analysis in this Case Study

The advantage of a dynamic approach is that found bugs are automatically reproducible and hence dynamic analysis a) leads directly to the problem and b) is sound, which means that it does not create false issues if no further approximations are made. Also, traces could be created in the regular life of the program. Consequently, found bugs are more closely to the real use of the library or program, instead of bugs in corner cases that are sometimes produced by other methods.

However, a basic requirement for the application of dynamic code analysis is that the analyst or analyzing software is able to run the code in its native environment. Yet, as our primary use case here was NASA space software and simulation or emulation software for the used operating systems and hardware configurations was not available, applying dynamic code analysis was out of our bounds.

4.2 Forward Symbolic Execution

In forward symbolic execution "instead of executing a program on a set of sample inputs, a program is 'symbolically' executed for a set of classes of inputs" [Kin76]. This results in creating constraints holding on different paths of execution. We explain the ideas of symbolic execution and describe its challenges and benefits. For further reference please refer to [CKP+11], which explains the state of the art and provides more detailed resources, especially of the various tools created recently.

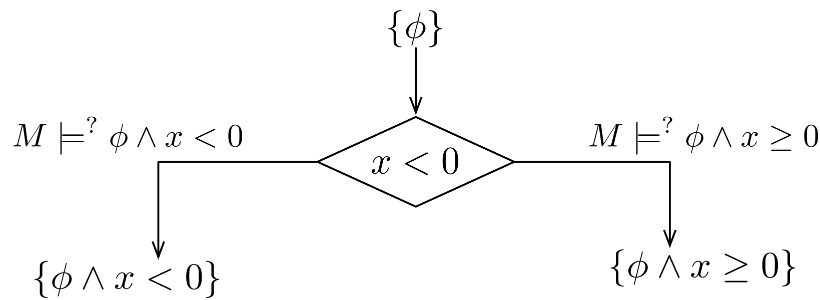

Figure 4.5: Forking in forward symbolic execution

4.2.1 Concept of Forward Symbolic Execution

In symbolic execution, instead of running a program with real values, the user defines a set of symbolic variables beforehand (e.g. an integer x). These values are unconstrained, meaning that in the theory of symbolic execution they can have any possible value. A specially designed virtual machine (or symbolic execution engine) then runs the code with these variables, recording all their manipulations. Whenever there is a fork in control flow (e.g. an `if`-statement $x < 0$?, see Figure 4.5) that depends on a symbolic variable, the virtual machine checks if, assuming the known constraints for the symbolic variables, both paths of the fork could be taken. Afterwards the constraint given through the branch (e.g. $x < 0$, $x \geq 0$ respectively) is added to symbolic variable. This way, the different forks create the various possible paths through the system, and on the path all constraints to the variables are recorded. Furthermore, whenever the code arrives at a critical point (e.g. a division by x), the machine checks if, given the known constraints on this path, the critical point may result in an error. In addition, symbolic execution can find unreachable code by walking all possible paths. Hence, if it can prove an `assert`-statement on all reachable paths, it can verify absence or presence of issues.

4.2.2 Applications of Forward Symbolic Execution

Symbolic execution has various applications. As open-source symbolic execution engines like KLEE [CDE08] are getting more and more mature, scientists extend these tools for experiments in various fields. Some of the applications are mentioned here; yet, new applications arise frequently.

Bug finding: At each critical instruction (arrays, pointer arithmetic, memcpy, etc.), the symbolic execution engine checks if this instruction can lead to a problem, given the path constraints computed. Critical instructions furthermore include reading and writing out of memory and division by zero. If a problem could occur, the engine generates and stores concrete inputs that lead to this bug. It can furthermore detect unreachable code, if the exploration of all possible paths terminates [CDE08].

Test generation: This is the most famous application of symbolic execution. Whenever a path through a program terminates (e.g. through an `exit` or the end of the `main` function), the machine creates a test case for this path. This is performed until the symbolic execution terminates (all paths have been walked) or until the user

terminates the process. If terminating, symbolic execution automatically generates tests with 100% branch coverage of all reachable branches [CDE08]. However, these are smoke tests, meaning that unless manually specified only runtime exceptions are checked. This form of testing does not check if the program follows a certain specification or generally does what it is supposed to do. Such checks could be implemented by defining the function's specification in `assert`-statements.

Exploit finding: Some approaches use symbolic execution for more subtle issues in code, namely possibilities to exploit bugs in software. For example, [ACHB11] analyzed open-source projects and successfully found various serious bugs. After determining the targeted part of code, their system even generates the exploit.

Differential symbolic execution: [PDEP08] apply symbolic execution on different versions of code pieces. Through symbolic execution they can determine semantic differences in pieces of code.

Functional equality analysis: Few work has been done to check arbitrary programs for functional equivalence, e.g. in [CDE08] or [RE11]. These works compare side-effect free functions, like standard C libraries against each other.

4.2.3 KLEE

As we were searching for an open tool that could check C code, KLEE [CDE08] was an obvious choice to meet our demands.

Features of KLEE

KLEE is an open-source symbolic execution engine that is developed at the University of Stanford. KLEE checks ANSI C code under Linux and is implemented in C++. It builds upon the LLVM compiler [LA04] and uses the constraint solver STP to calculate feasibility of paths. "STP is a decision procedure for the satisfiability of quantifier-free formulas in the theory of bit-vectors and arrays that has been optimized for large problems encountered in software analysis applications" [GD07]. KLEE uses various optimizations for speeding up the search, decreasing memory requirements and increasing code coverage [CDE08]. Instead of variables, space of memory is marked as symbolic (*bit-level accuracy*), which increases the applications to more complex data, e.g. strings. Based upon these symbolic variables and the program code, tests are generated and statement and branch coverage is printed. Furthermore native execution of tests, including linking to gcov [Fre12] is possible to verify the just mentioned code coverage values as well as reported issues.

KLEE is able to detect various bugs in the code: Out of the box, KLEE can find issues by falsifying `assert`-statements under given constraints, looking for buffer overflows and reading out of bounds, as well as runtime issues like division by zero or unreachable code [CDE08].

1st step:

```
char localPathP[100];
klee_make_symbolic(localPathP, (sizeof localPathP), "
    localPathP");
```

2nd step:

```
klee_assume(localPathP[100-1] == '\0');
```

3rd step:

```
assert(OS_API_Init(localPathP) == 0);
```

4th step:

```
(write models in C code)
```

5th step:

```
llvm-gcc --emit-llvm -c \
    -I../../include/ -g -Wall -pedantic -Wstrict-
        prototypes -fprofile-arcs -ftest-coverage -
        DOS_DEBUG_LEVEL=3 -DINTERCEPT -D_EL -DENDIAN=_EL -
        DSOFTWARE_LITTLE_BIT_ORDER -D__ix86__ -D_ix86_ -
        D_LINUX_OS_ -Dposix -DX86PC -DBUILD= -D_REENTRANT
        -D_EMBED_ \
    OSFS.c
```

6th step:

```
klee --optimize --allow-external-sym-calls --libc=uclibc
    --use-cex-cache --only-output-states-covering-new --
    output-stats \
    ./OSFS.o
```

7th step:

```
KLEE: output directory = "klee-out-147
KLEE: done: total instructions = 19169
KLEE: done: completed paths = 101
KLEE: done: generated tests = 5
```

8th step:

```
ktest file : 'test000001.ktest'
args       : ['./OSFS.o']
num objects: 7
object    4: name: 'localPathP'
object    4: size: 100
object    4: data: '\x01\x01\x01\x01\x01\x01\x01\x01\x01\
    x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x02\x01\
    x01\x01\x01\x01\x01\x01\x01\x01\x02\x01\x01\x01\x01\
    x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x01\x02'

Error: memory error: out of bound pointer
File: /home/fraunhofer/klee/examples/osal/model.c
Line: 113
Stack:
        #0 00000697 in __user_main (numArgs=1) at /home/
            fraunhofer/klee/examples/osal/model.c:113
        #1 00005188 in main (=1, =50531248)
```

9th step:

```
gcc -I../../include/ %[...]
    /home/fraunhofer/klee/Release+Asserts/lib/
        libkleeRuntest.so \
    OSFS.c -o OSFS.out
replay-ktest:
    KTEST_FILE=$(FILE) ./OSFS.out
replay-run-all-tests:
    for file in $(TEST_FILES); \
        do make replay-ktest FILE=$$file; \
    done
```

Figure 4.6: The KLEE process

KLEE Process

When using KLEE, the analyst can follow a pretty straightforward process (see Figure 4.6), which consists of a few basic steps:

1. Define the variables or memory locations that are supposed to be unconstrained, i.e. symbolic. This choice is vital to the performance and resulting coverage. One can achieve this with the call to the KLEE function `klee_make_symbolic`, providing the address and size of the memory, which is supposed to be symbolic, as well as the name of this variable for debugging information.
2. Sometimes we do not want to check the full range of possibilities for certain variables. This is especially true when dealing with strings, where we usually assume that strings are terminated with a `'\0'`. In practice we add constraints to the variable that was unconstrained before. This step also includes creating a valid symbolic state of the system by creating and initializing global variables.
3. If certain properties should be checked, we can encode them into `assert` statements.
4. In order to be able to check software with KLEE properly, we need to compile the code and all referenced libraries with LLVM. This means that source code of all libraries to be used with symbolic variables must be available, as KLEE will execute these functions symbolically as soon as a symbolic variable is used as parameter. KLEE can be configured to use a concrete value of a variable if source code is not available. However, the advantages of symbolic execution usually blow up in smoke when the function is executed with a concrete value. Therefore, usually external libraries or system calls need to be mocked to properly analyze a system. The developer creates mocks with short implementations of the functions and replaces the original function calls. The replacement can be performed with preprocessor macros.
5. As soon as external references are created, we can compile the whole system with LLVM. Regular preprocessor macros may be handed to the compiler as usual.
6. After these steps we can run KLEE on the created LLVM code. KLEE provides plenty of options, which are mostly not very well documented. As these options can have a big impact on both speed of execution and code coverage, testing and analyzing the various options is an important yet annoying task.
7. KLEE informs whether the run has terminated and gives some statistics.
8. After the run we can look into the tests and reports created to analyze the concrete issues found. These bug reports can be a little cryptic as memory is output in its byte representation.
9. (Optional) If we want to double-check KLEE's results, we can rerun the tests provided natively afterwards. KLEE can also be configured to support gcov [Fre12], a widely used code coverage tool, which gives deeper analysis of reached lines and branches. Upon unsuccessful runs by KLEE, this tool is very useful to determine which lines remained untouched by the symbolic execution process.

4.2.4 Challenges of Symbolic Execution

Symbolic execution is a very powerful approach. However, it faces three big bottlenecks:

State explosion: At every branch in the control flow of a program, symbolic execution takes all possible paths. Accordingly, the number of states rises exponentially with the number of branches, even when not all branches lead to a forking of the control flow. This state explosion is a problem of both computing all paths but also storing all the constraints of each possible path that is not yet completely executed. Various symbolic execution tools tackle the latter issue by sharing the states between different paths or pruning redundant paths. Both issues are addressed by parallelization through cloud computing, e.g. with Cloud9 [CZB+10].

Complex constraints: Complex, real world applications need to solve non-linear mathematical problems, as these are usually approached in the code. These computations need to be performed by the constraint solver. Even though constraint solvers made big progress recently, new heuristics are needed to address this problem [CKP+11].

The environment problem: Lastly, the environment problem relates to interactions of the program with everything outside the pure code. This includes libraries and everything that is related to the operating system, such as file systems or multi-threading.

4.2.5 Symbolic Execution for Equivalence Analysis

The question is, how this can be used for equivalence analysis. Theoretically speaking, symbolic execution can create a set of pre and post conditions, which are representations of the semantics of a program. Hence, in order to compare semantics, we can compare the pre and post conditions calculated by symbolic execution. In practice we can just use the `assert`-statement and compare the return values of the two functions called. For our more complicated use case, some more work needs to be done, which we will explain in the following sections.

The idea of comparing implementations was first described in [CDE08] and further developed in a work parallel to this one, in [RE11]. However, the named sources describe very basic examples. The case studies describe very loosely coupled functions, i.e. functions with few interactions and dependencies. Furthermore, the functions are independent from a global state, without side effects. Additionally, deeper knowledge of system calls and operating systems is not required, as the examples only analyze different software libraries running on the same machine. Nevertheless, the results are quite promising.

To explain this in more detail the example from [CDE08] is given in Figure 1. In this example two functions calculate the modulo operation, i.e. the remainder of a division. The topmost implementation (`mod_opt`) is an optimization for numbers that are powers of two, whereas the second function (`mod`) implements the native modulo operation. We would like to check then whether the two implementations have the same functional result. The symbolic execution is initialized in the main function, by marking the two input variables x and y as symbolic. We have no constraints to the input values here, nor any external function calls. Hence, we can directly compile the program and execute it with KLEE.

```
unsigned mod_opt(unsigned x, unsigned y){
    if ((y & -y) == y){ // Is y a power of two?
        return x & (y-1);
    }else{
        return x % y;
    }
}
unsigned mod (unsigned x, unsigned y){
    return x % y;
}
int main(){
    unsigned x,y;
    klee_make_symbolic(&x, sizeof(x));
    klee_make_symbolic(&y, sizeof(y));
    assert(mod(x,y) == mod_opt(x,y));
    return 0;
}
```

Listing 1: Return value equivalence in a simple environment with KLEE, from [CDE08]

KLEE will create the two symbolic variables and will jump into the `mod` implementations first. It will realize that y must not be zero and create a test case for this issue, furthermore assuming $y \neq 0$. Afterwards it will store the result of the call ($x \mod y$), and jump into the `mod_opt` implementation. At the `if`-clause KLEE will check the condition and jump into two paths: One the one side KLEE checks if $y\& - y == y$ is contradictory under the current path constraints, which is not the case as the constraints are empty. If it was contradictory, KLEE would recognize that this path cannot be walked and take the other path. In the present case however, it will branch the path and add $y\& - y == y$ to the path constraints. Inside the `then`-clause of the `if` it will return $x\&(y-1)$ to the `assert`-statement. KLEE now asks its constraint solver STP if the assert can fail under the current path constraints, namely $x \mod y == x\&(y-1)$, assuming that $y\& - y == y \wedge y \neq 0$ holds. STP will tell KLEE that it always holds, the path terminates at the `return 0` and the symbolic execution backtracks to the branch at the `if`-clause. The `then`-block is not very spectacular, KLEE assumes that $\neg(y\& - y == y)$ holds and returns $x \mod y$. Again, it asks STP to falsify $x \mod y == x \mod y$ under the assumption of $\neg(y\& - y == y) \wedge \neq 0$, which is not possible. Accordingly, KLEE will quit with three different test cases: $y = 0$, y is a power of 2 and y is not a power of 2. As all possible paths were taken and the `assert`-statement failed on none, KLEE verified that the two implementations always return the same value.

4.2.6 Leveraging Symbolic Execution for Real Equivalences

In order to check proper equivalences of real world code, some extensions had to be made to the example.

C standard library and POSIX environment: In order to check code with KLEE, we need to compile it properly with LLVM, which requires that all libraries are compiled

```
int main(int cArgs, char* args[]){
    int pre;
    int sideeffect1;
    int sideeffect2;
    char* res1;
    char* res2;

    klee_make_symbolic(&j, sizeof(int), "j");
    klee_make_symbolic(&globalVariable, sizeof(int), "
       sideEffect");

    /* preserve system state */
    pre = globalVariable;

    res1 = test1(j);
    sideeffect1 = globalVariable;

    /* resume system state*/
    globalVariable = pre;

    res2 = test2(j);
    sideeffect2 = globalVariable;

    /* compare system states*/
    klee_assert(res1 == res2);
    klee_assert(sideeffect1 == sideeffect2);
}
```

Listing 2: Excerpt of checking side-effects with KLEE, full version in Appendix D.1

with LLVM as well. This is not the case for the common standard C library as well as for POSIX system calls. However, KLEE offers functionality for some POSIX and standard C calls through the uClibc. Yet this functionality is not complete. We had to create various models to properly simulate the real environment for our use case.

Side effect equivalence: To find equivalence as defined in Section 3.2.2, we need to determine side effect differences. We check this through `assert`-statements similar to those demonstrated in the example in Listing 2. To do this, we extract all global variables that are used by the function and store their values before and after the call of tested functions. The new values of the global variables are stored in the `sideeffect` variable. After calling both functions, we check the results of the variables as well as the side effects against each other. The long version of the code can be found in Appendix D.1.

Reset of side effects: It is important to note that KLEE does not execute the functions in parallel. Both functions are called one after another, as described in the example in the last section. This means that we have to prevent functions having an impact on each other, by determining and reverting existing side effects. In our solution, the global variables are reset with the stored values after the first function call.

Setup of a consistent global state: Lastly, in stateful applications, we also need to create a consistent global state. This means that a proper setup of the symbolic variables needs to be created. In the example added in Appendix D.2 a state machine is created that manipulates the code depending on a certain set of variables. In order to check the code with high coverage, we need to ensure that the global state (i.e. the current state of the state machine) is set to a properly constrained symbolic variable. If we would initialize it with a certain concrete state, the symbolic execution would only check one step of the state machine. In a different situation, when dealing with structures and arrays, we must make sure that dependencies within arrays are kept in the symbolic variables. For example, when creating a structure that always contains two integers with an dependency, say a number and its square root, when marking the first integer as a symbolic variable, we have to keep the dependency intact, i.e. create a constraint that the second value is the square root of the first. Otherwise we generate false positives that do not appear in regular execution of the software.

Please be aware that the comparison runs on the same machine, compiled in LLVM. This implies the assumption that the code in the original OS, on the original hardware with the original compiler, is semantically equal to the version compiled with the LLVM compiler in the environment of the operating system running the analysis. This assumes that bugs in both compilers do not affect the executed code as well as the code does not depend on certain hardware. This is especially important when code directly references to certain hardware configuration, for example preprocessor macros that checks $x86$ or $x64$ bit environment. If such references appear, we have to make sure that the system analyzing the software has the same configuration than the environment running the software.

4.2.7 Successful Applications

We applied symbolic execution to a function in the Operating System Abstraction Layer (OSAL, see Section 8) project. We chose `OS_TranslatePath`, the function that had the least connection to the environment and was the most self-containing, although it still contains calls to the Standard C Library and various global variables (the functions are listed in Appendix E.1, the POSIX implementations is renamed to `OS_TranslatePathP`, for the RTEMS version see `OS_TranslatePathR`).

We checked for side effects on the parameters with the KLEE script listed in Appendix D.3. As a result we found one issue in the OSAL project that was present in all implementations (see Figure 4.7). It led to the system reading a piece of memory out of the bounds of the targeted array. This can lead to unexpected behavior.

```
/*
** We want to find the number of chars to where the second "/" is.
** Since we know the first one is in spot 0, we start looking at 1, and go until
** we find it.
*/
NumChars = 1;
while ((VirtualPath[NumChars] != '/') && (NumChars <= strlen(VirtualPath)))
{
    NumChars++;
}
```

Figure 4.7: A buggy piece of code discovered with KLEE

The fact that the bug was present in all implementations shows a huge benefit of symbolic execution. It contains some rules of absolute correctness, in contrary to approaches based on comparison of implementations. Thus, bugs like the one demonstrated can be found, even if present in all versions. The downside is the time it takes to set up the test case. Maybe automation can help extracting the relevant symbolic variables, the models to create for external function calls and other preparations needed.

4.2.8 Conclusion of Applying Symbolic Execution for Equivalence Analysis of SALs

Applying a symbolic execution tool like KLEE for equivalence analysis of SALs is generally spoken a very exact method to find real runtime errors. It is especially good for finding issues in extreme corner cases, as demonstrated in [CDE08]. As a bonus it can find certain kinds of errors that are present in all implementations. This goes beyond the method of comparing implementations for pure functional differences. However, some effort has to be undertaken to analyze real equivalence, which might sometimes be a bit more complicated, as side effects of a method have to be determined. Furthermore one has to determine the best way to revert the side effects of the function. We proposed an adequate method for our use case presented above.

SALs are created in order to deal with differences in technology. Differences vary from slight deviance between implementations such as in the implementations of the POSIX API to strong differences such as whole middleware layers. Sometimes SALs provide

implementations for different environments. As such references to the environment targeted are very common in these implementations. Accordingly, these environments need to be modeled for symbolic execution in order to put the state explosion problem on a lead. However, as models are abstractions of the real behavior, little differences are introduced. As symbolic execution is a sound tool, it will find these little differences introduced when searching for other differences, hereby destroying the soundness of the tool, which is one of its main advantages. In other words: If the tool reports something, either the code or the model is broken - but usually it is the model. This does not drive the tool useless (we found a bug during application), but affects the results strongly. When considering the amount of work to be invested in order to build the appropriate models for checking the environment-intense code of OSAL, and knowing on the other side that differences might result from difference in system or library calls, or compilers and hardware setup under deployment, we come to the conclusion that symbolic execution is more successful in self-contained environments than in our use cases.

4.3 Static Code Analysis

The third method to determine functional equivalence for SALs we would like to present, is static (code) analysis. "Static analysis involves analyzing the source code of a module at compile time to find or ensure the lack of a certain class of defects" [LVH10]. Advantages of static analysis include:

- Controllability: In comparison to dynamic analysis it is usually easy to apply static analysis, as it usually does not require integration into the "natural" habitat of the software. This enables analyzing code of various platforms and/or hardware settings without being in possession of the whole runtime environment [LVH10].

- Coverage: When a defect like naming conventions or lack of initialization is clearly specified, static code analysis tools can often check the whole system against the defect. As static analysis usually checks the code instead of executing its paths, the problem can be approached faster and more completely.

Yet, static analysis cannot be complete (find all issues) and sound (make no mistakes) in finding semantic properties of a program. Consequently, many static analysis approaches suffer from the fact that proposals of many defects turn out to be wrong [Bin07]. This is a problem that we face with machine learning techniques, which will be explained in Chapter 6. Furthermore, as soon as semantic analysis of the code is performed, e.g. deep analysis of loops, static analysis reaches the same issues as other approaches such as symbolic execution, namely incompleteness and non-termination. This is again due to the fact that we cannot certainly decide any non-trivial property for all programs (see Section 3.2.2)

However, the advantages were more convincing for our use case. Consequently we decided to apply static analysis techniques. Two different static approaches for determining software equivalence came to our mind: bug patterns and reviewing.

4.3.1 Bug Patterns

The first way to approach equivalence analysis with static analysis is through bug patterns. Bug patterns are definitions of possible flaws in program code, such as unreachable code or possible access to uninitialized variables. With good tools and large sets of error patterns tools like FindBugs [CHH+06] or Understand [Too12], static analysis is both easily applicable and can be successful in avoiding common problems of programming.

However, this approach also has two disadvantages.

- Bug patterns must first be written so that bugs can be found. This leads to the existing libraries of issues for regular static analysis, such as [Hat04] or others. However, in contrary to performing reviews, everything that is not yet specified and written down cannot be found. This means that the so-called unknown-unknowns are out of scope for the tool.
- Further information from subsystems cannot be easily acquired. Real life software is usually connected to several subsystems and libraries. In order to understand the semantics of the software, the semantics of subsystems are required. For machines, this is again a complex task, especially as developers often do not specify their code in a machine-readable language.

4.3.2 Manual Reviews

The second approach is based upon the human understanding instead of predefined machine-coded knowledge. During a reviewing process one or more analysts read through code with a certain goal. The goal of reviewing two functions for equivalence analysis is to determine if semantic differences exist (and where they are located), whether these differences are part of the abstracted aspect (see Section 3.2.2) and consequently if the functions are equivalent.

Reviewing has been shown to be a successful technique for improving code quality [PJ97]. Especially its structured and more formal pendant, the so-called Fagan inspection [Fag76] optimizes its effectiveness for defect removal. Furthermore, code reviews have additional benefits for the team, such as improvement of communication or clarification of requirements [PJ97].

Conducting manual reviews can have various advantages over automated approaches: Human beings can often quickly identify semantics of programs, which is an undecidable task for computers (see Section 3.2.2). If deeper knowledge of the code is required, for example semantics of the referenced libraries, it can be easily accessed through the web. In addition to the advantages for finding functional equivalence, manual reviews can furthermore increase the communication and knowledge transfer in the team. As the team discusses the code, it can help identifying differences in code style and architecture.

However, reviews can be costly through the invested human effort. Supporters of reviewing and Fagan inspections usually justify this with the previously mentioned benefits (as in [PJ97]). Furthermore, manual reviews are threatened by missing subtle details. In contrary, a mechanic tool that is designed to identify these differences will miss a difference only if it reaches the boundaries of computation or if there is a bug in the tool itself.

4.3.3 Tool-Supported Reviewing

When analyzing advantages and disadvantages of both approaches, it turns out that a combination of the two is needed. Other research in static analysis comes to the same conclusion: "The tasks that machines tend to be good at are tedious bookkeeping tasks. In contrast, programmers are better able to make 'ah ha' type discoveries. This suggests the use of semi-automatic analysis where peak performance is obtained through a symbiotic relationship between the two." [Bin07]

Consequently, we created a prototype that combines the advantages of each method in a three-step approach (see Figure 4.8):

1. At first, Static analysis identifies differences in code. Here the computer shows its strength in finding even the smallest differences in the source code. This step is explained in the next chapter.

2. Afterwards, machine learning applies filtering. As mentioned previously, static analysis often faces issues with too many false positives. Existing machine learning techniques help for this step. We describe our approach towards this issue in Chapter 6.

3. Lastly, human analysts understand the code and identify functional equivalence or code issues, which will be described in Chapter 7.

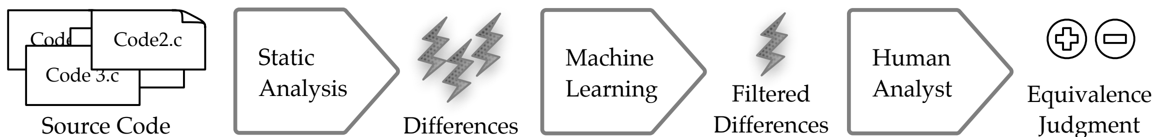

Figure 4.8: Overview of our approach

Part III

Tool Support for Reviewing Software Equivalence

Chapter 5

Detecting Differences in Code

"Source code analysis is the process of extracting information about a program from its source ...using automatic tools" [Bin07]. In order to do this, the code has to be processed and analyzed with certain goals. The first section describes the automatic tool we designed. It explains how the information is detected in the analyzed program. The second section describes which information is extracted from the source code.

5.1 Process of Detecting Differences

The analysis of source code is split up into three parts (see Figure 5.1): First, processing the various code files into analyzable function pairs. Second, retrieving context information for the analysis from related resources, such as libraries or header files, and third the analysis of the code itself.

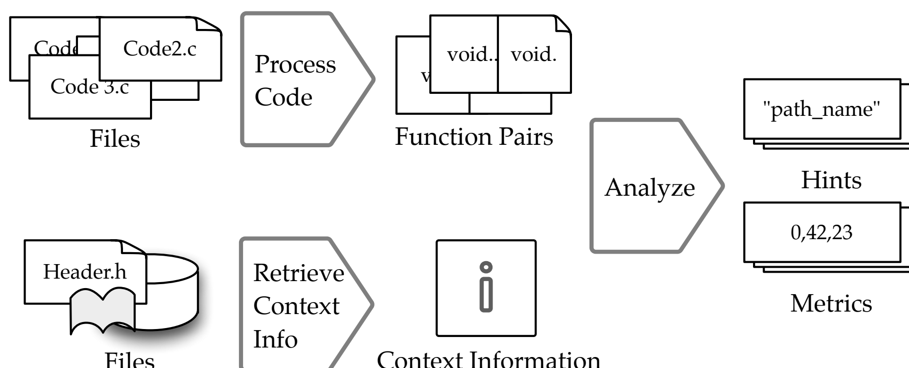

Figure 5.1: Static analysis process

Processing Code

Processing the code files in order to create analyzable pairs of functions comprises five steps, which can be seen in Figure 5.2.

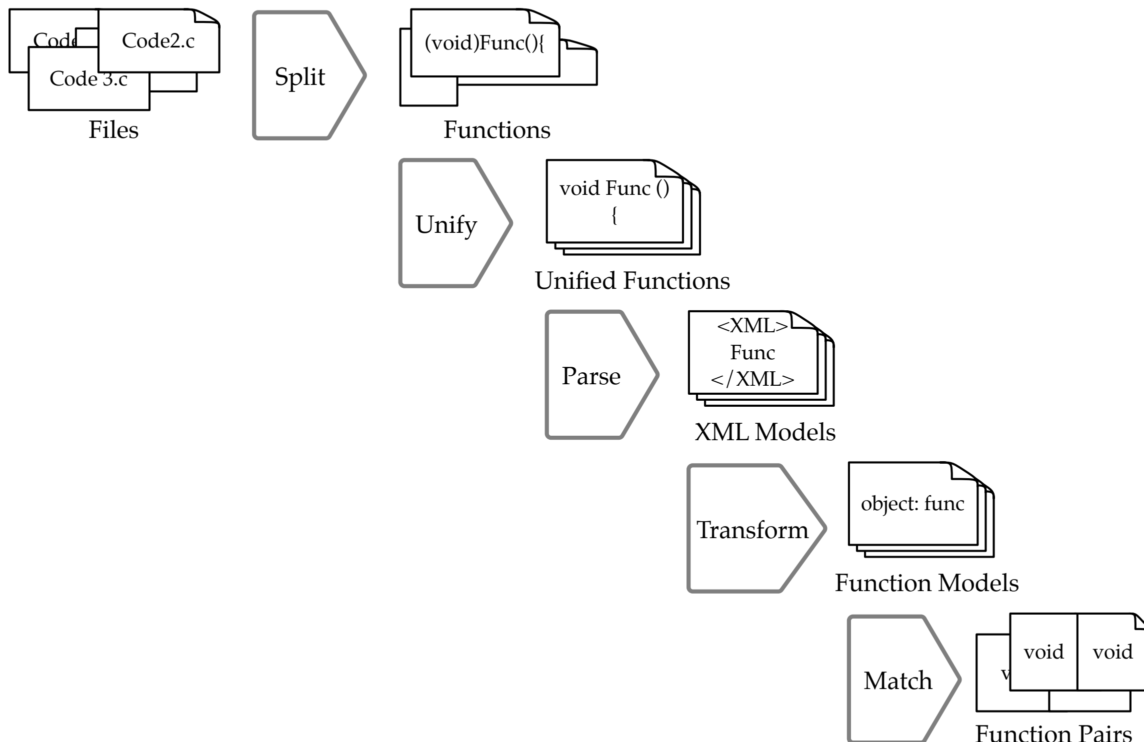

Figure 5.2: Processing of code

Splitting up files: The code is usually available in form of source code files. As discussed in Section 3.3, we want to analyze code on the level of functions. Therefore, we need to split up the files into functions.

Unifying functions: Some differences in source code are a matter of code style. In order to prevent code style from having an influence on the analysis, we eliminate it as much as possible with the *uncrustify* tool [Ope12c]. We used the options listed in Appendix C.1.

Parsing functions: Afterwards functions are parsed into an XML representation with the srcML tool [MC04]. The output is an XML representation of the abstract syntax tree, which makes the code easier to process and analyze.

Transforming functions: Next, the XML code is converted into the model used for static analysis. The model is a data representation of the real code, containing classes for functions, statements, variables etc. In order to properly analyze variables, we needed to write a new parser for the expressions used. During this step reappearing variables and other code elements are identified. For example, if a variable is defined

at the beginning and appears in a statement afterwards, it is looked up in the namespace and referenced properly. After this step, the function is transformed into an analyzable, linked representation of the abstract syntax tree.

Matching functions: Lastly, we take the single functions and match them to have pairs of different implementations of a function. Matching is performed based upon the signature.

Retrieve Context Information

In addition to the code itself, sometimes further information is useful to analyze a function. For example, we might want to know if a called function is defined within the system under analysis, or we want to check the type of a certain variable that is not defined within the current function.

The library created for our approach contains a flexible context model that can contain all sorts of information, which is easily accessible through a dictionary. In practice we added two kinds of information: header files and an index of functions from the C standard library. The former serves as an index of functions of the system under analysis, the latter serves as a reference to check if the function is a standard library function.

Analyze the code

With the information from the context and the model of the code, we can now perform the actual task: analyzing the code. In order to do this, we created components that extract information from the code, the so-called *data extractors*. The library implemented in this thesis is designed to enable easy extension of data extraction. Hence, additional data extractors can be defined easily and added to the tool.

The result of this data extraction has two forms: *hints* and *metrics*. Hints are concrete source code elements that indicate differences (such as variable names) whereas metrics are numerical representations of these differences. For example, when searching through the usage of variables and finding one variable that is not used in one of the implementations, the extractor could return the name of the variable as a hint and quantify the metrical difference with 1. In the following section we will explain the data extractors developed and used within this thesis.

5.2 The Data Extractors

We wrote data extractors for various forms of differences. Each of the 16 extractors focuses on one possible type of variance in the code. The differences were of eight distinct types.

- Return code difference
- Side effect difference
- Function call difference
- Variable difference
- Value difference
- Lines of code difference

- Levenshtein distance
- Cyclomatic-complexity difference
- Operator difference

We describe these in detail in the following sections.

Return Code Difference

Return code difference checks the return value of implementations. Of course, this is again an undecidable problem, as it asks for the semantics of a program. Hence we have to apply heuristics. In the present use case, return codes are used to communicate success or failure of functions; hence, the return statements usually contain either only a constant or a variable that is used to store the return value. Hence, we wrote a data extractor that isolates all return values, and, if the return value is a variable, the different values assigned to that variable. The extractor is calculated by the number of elements that are only part of one of the implementations, the symmetric difference, which is defined as

$$\mathrm{SymmDiff}(\mathrm{Set}_1, \mathrm{Set}_2) = (\mathrm{Set}_1 \backslash \mathrm{Set}_2) \cup (\mathrm{Set}_2 \backslash \mathrm{Set}_1)$$

Hence, the data extractor is defined like following:

Return code difference	
Hints	$\mathrm{SymmDiff}(RC_{\mathrm{Impl1}}, RC_{\mathrm{Impl2}})$
Metric	$\lvert\mathrm{SymmDiff}(RC_{\mathrm{Impl1}}, RC_{\mathrm{Impl2}})\rvert$

where RC_{Impl1} and RC_{Impl2} are possible return codes of the implementations Impl1 and Impl2.

Side Effect Difference

Following the definition from Chapter 3.2.2, not only return codes, but also side effects are relevant to functional equivalence. Hence we added a heuristic that determines which external variables (parameters passed through call-by-reference or global variables) could possibly be altered by a function.

Again we calculate the extractor with the symmetric difference of the set of parameters that is written to at some point of the respective functions. For the global variables we determine which variables are manipulated, remove those that are defined as a parameter or within the function and again look for variables that show up only in one of the two implementations.

The first data extractor relates to side effects by writing of global variables.

Global writing difference	
Hints	$\mathrm{SymmDiff}(GW_{\mathrm{Impl1}}, GW_{\mathrm{Impl2}})$
Metric	$\lvert\mathrm{SymmDiff}(GW_{\mathrm{Impl1}}, GW_{\mathrm{Impl2}})\rvert$

where GW_{Impl} gives all variables that are changed within the function Impl and are not listed in the parameters or defined in the function.

The second data extractor checks side effects by manipulating parameters.

Parameter writing difference	
Hints	$\text{SymmDiff}(PW_{\text{Impl1}}, PW_{\text{Impl2}})$
Metric	$\lvert\text{SymmDiff}(PW_{\text{Impl1}}, PW_{\text{Impl2}})\rvert$

where PW_{Impl} is a list of variables that are manipulated within the function Impl.

Function Call Difference

We knew from our case study that differences in function calls do not necessarily reflect semantic differences (for an example see Figure 8.4). We split up function calls into three different groups, in order to differentiate if the various implementations have access to the functions.

Global Function Calls are calls to those functions that are definitely available in all implementations. If the implementations are instances of an API then calls to functions defined within this API are always global function calls.

Local Function Calls are calls to functions that are only available to a specific implementation. If the implementations are deployed on different systems the system calls are local to the respective system, hence, local function calls.

Local Global Function Calls are function calls that could be shared between implementations. Libraries that may be included in the implementations are part of this domain. The local and local global function calls are special in the sense that the code called is out of scope of the implementation.

We created a list with standard library functions and search for function definitions of the API in header files provided. Hence, we classify all calls to functions from the header files as global function calls, because these functions are definitely visible and available to all functions. We furthermore classify all calls to the standard library as local global function calls, as those are possibly globally visible, but implementations might differ, as they are local to the compiler. Third, we categorize all other function calls as local function calls. These are calls that are probably only related to the respective system, such as system calls.

The three different extractors are again calculated using the symmetric difference.

Function call difference of domain D	
Hints_D	$\text{SymmDiff}(FC(\text{Impl1}, D), FC(\text{Impl2}, D))$
Metric_D	$\lvert\text{SymmDiff}(FC(\text{Impl1}, D), FC(\text{Impl2}, D)\rvert$

For simplicity we list the three extractors in one table. $FC(\text{Impl}, D)$ gives all functions of domain $D \in \{\text{local}, \text{global}, \text{local-global}\}$ called within Impl.

Variable Usage Difference

Next we found that sometimes developers use varying constants in their code. We explore this by determining read access to the variables of a function. Every access to a variable is extracted from the function and deviance between implementations is calculated. We furthermore created two additional extractors for constant variables: all constants and project constants, which are constants that are defined within the system under analysis. Both are classified through regular expressions: By default upper-case variables are guessed as constants, and a certain prefix is defined for the project constants (e.g. in OSAL all constants are prefixed with `OS_`). As usually, difference is calculated through mutually exclusive read access to variables.

Variable usage difference	
Hints	$\text{SymmDiff}(RV_{\text{Impl1}}, RV_{\text{Impl2}})$
Metric	$\lvert\text{SymmDiff}(RV_{\text{Impl1}}, RV_{\text{Impl2}})\rvert$

where RV_{Impl} are the variables read by function Impl.

Constant usage difference	
Hints	$\text{SymmDiff}(RC_{\text{Impl1}}, RC_{\text{Impl2}})$
Metric	$\lvert\text{SymmDiff}(RC_{\text{Impl1}}, RC_{\text{Impl2}})\rvert$

where RC_{Impl} are the constants read by function Impl.

Project constant usage difference	
Hints	$\text{SymmDiff}(RPC_{\text{Impl1}}, RPC_{\text{Impl2}})$
Metric	$\lvert\text{SymmDiff}(RPC_{\text{Impl1}}, RPC_{\text{Impl2}})\rvert$

where RPC_{Impl} are the project constants read by function Impl.

Value Difference

Furthermore, we figured that not only constants but also constant values could show semantic differences. For example `for`-loops might iterate a different number of times. These values can also serve as an indicator for magic numbers in the code. Accordingly, we extracted all string and integer values used in the implementations and created hints and metrics for these two categories.

String value difference	
Hints	$\text{SymmDiff}(SV_{\text{Impl1}}, SV_{\text{Impl2}})$
Metric	$\lvert\text{SymmDiff}(SV_{\text{Impl1}}, SV_{\text{Impl2}})\rvert$

where SV_{Impl} are all strings defined in the function Impl.

Integer value difference	
Hints	$\text{SymmDiff}(IV_{\text{Impl1}}, IV_{\text{Impl2}})$
Metric	$\lvert\text{SymmDiff}(IV_{\text{Impl1}}, IV_{\text{Impl2}})\rvert$

where IV_{Impl} are all integer values used in the function Impl.

Lines of Code Difference

The lines of code difference is a very rough metric. It takes the statement lines of code, i.e. the number of semi-colons in the C code, and calculates the difference. As such it only calculates a metric and no hint.

LoC difference	
Metric	$\lvert SLoC_{\text{Impl1}} - SLoC_{\text{Impl2}}\rvert$

Levenshtein Editing Distance

Editing distance determines how many text changes (adding, changing, deleting characters) have to be made to convert one text into another. It was first suggested in [Lev66].

To give an example, the editing distance of "pale" and "nail" is 3. It needs three steps to get from one word to the other: pale - nale - nal - nail.

This extractor is useful in order to roughly see how different two pieces of code are. We used the implementation from [Ope12a]. However, this extractor does not produce concrete hints, only a metric.

Levenshtein distance	
Metric	$Lev_{\text{Impl1,Impl2}}$

where $Lev_{\text{Impl1,Impl2}}$ denotes the Levenshtein distance between Impl1 and Impl2.

Cyclomatic-Complexity Difference

Next we experimented with differences in control flow elements, such as `if`-statements or `for`-loops. We ended up integrating a metric that sums up these differences: the cyclomatic complexity, which counts the number of independent paths that can be walked through the control flow graph (CFG). McCabe presented the approach in [McC76].

McCabe complexity is calculated based on the CFG:

$$\text{McCabe}_p = E_p - N_p + 2P_p$$

where E_p is the number of edges in the CFG of p, N_p is the number of nodes, and P_p represents the number of connected components. Connected components are subgraphs where each node of the component is connected to all other nodes of the component through one or more paths, but is not connected to any other node of the graph outside the component.

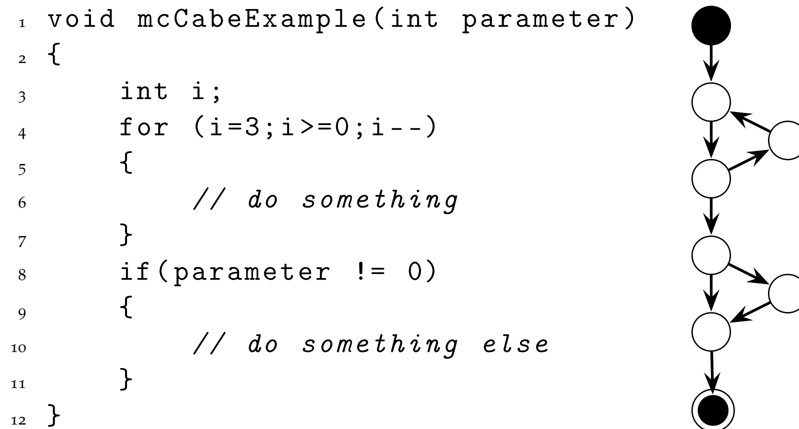

Figure 5.3: A code example and its CFG

In the example in Figure 5.3 we wrote a short program containing a `for`-loop and an `if`-branch. The CFG of the example contains 9 nodes, 8 edges and 1 connected component (the whole program). Accordingly, the cyclomatic complexity of the program is:

$$\text{McCabe}_{p5.3} = 9 - 8 + (2 \times 1) = 3$$

In our tool, we used the library provided by [Blu12] with the options to be found in Appendix C.2. For the metric, we used the difference of the cyclomatic complexity of two programs.

This extractor creates only a metric and no hints, similarly to the previous extractor.

Cyclomatic-complexity difference	
Metric	$\lvert\text{McCabe}_{\text{Impl1}} - \text{McCabe}_{\text{Impl2}}\rvert$

Operator Difference

Lastly we added a third group of features: Operator differences. During the analysis we found that various implementations were confusing the boundaries of loops, which will be discussed in detail in Section 8.4.1. Hence, the operators could make a difference in equivalence analysis.

However, there are various operators for all kinds of expressions: Arithmetical operators that compare numbers, operators that calculate integers or operators for boolean variables. Therefore, we created four different extractors, which count the appearing symbols in the code (see Table 5.1). The hint is the symbol itself and the metric is calculated by the sum over all differences of appearance of the symbols

$$\text{OpDiff}(p_1, p_2, S) = \sum_{s \in S} \lvert\text{Count}(s, p_1) - \text{Count}(s, p_2)\rvert$$

where S is the set of symbols and $\text{Count}(s, p)$ counts the number of times s appears in p.

Feature	Symbols counted
Operator Diffs	All
Comparison Operator Diffs	$\{<=, >=, ==, !=, <, >\}$
Arithmetic Operator Diffs	$\{++, --, +, -, /, *, \wedge, \%\}$
Logical Operator Diffs	$\{\&\&, \vert\vert, \&, \vert, !\}$

Table 5.1: Operator sets for data extraction

	Operator difference of operator set S
Hints	$\{s \vert s \in S \wedge \vert \text{Count}(s, \text{Impl1}) - \text{Count}(s, \text{Impl2}) \vert > 0\}$
Metric	$\vert \text{OpDiff}(p_1, p_2, S) \vert$

where S is a set of operators as shown in Table 5.1 and Count and OpDiff as defined above.

5.3 Summary

In this section we used static analysis to determine several forms of deviance between implementations. We calculated hints, which are possible spots of difference and their numerical significance, the metrics. However, due to large numbers of differences in code, big quantities of hints and metrics exist. We will determine ways to extract the relevant differences in the next two chapters.

Chapter 6

Classification and Filtering of Hints

As static analysis returns a huge number of differences, two systems would be of great help:
1. An automation mechanism prioritizing the various function pairs to compare. Maybe the mechanism can already give a guess whether two functions are equivalent or not. Accordingly, the user could focus on the function pairs where the system was uncertain.
2. An automation mechanism that could understand which of the hints provided by the static analysis are relevant and which hints are probably not relevant to the user.

In order to explain the strategies we created to solve these problems, we will first introduce into the terminology[1] used in machine learning, and explain our approaches afterwards.

6.1 Terminology

For determining whether a function pair is equivalent or not, we need a *classifier* (or *prediction model*), an automation mechanism that learns from a set of known examples how to classify unseen examples [WFH11, page 40]. An *example* or *instance* is one set of data, in our case the data of one function pair. All examples together form the *example set*, which represents the input to our classifier. The output of the classifier is called the *label*, which can be a binary result, an item out of a set of items, or also a real number.

In order to classify examples, classifiers usually require not the instances themselves, but *features* thereof (sometimes also called *attributes*). These features are data points that describe the instance. Accordingly, each instance can be described as a vector $A \in \mathbb{D}^N$ with features a_i for $i \in [1, N]$ with N being the number of features. The domain $\mathbb{D}$ can

[1] Please note that explaining the whole state of the art would go beyond the scope of this work. Accordingly, only the applied technology is explained. Several extensions and variations of the applied approaches exist and might be interesting to analyze in future work.

be numerical or nominal. In our case the label is a nominal type of one of the values *Equivalent* or *Different*. Usually the set of features contains various items that may or may not be useful for classifying the label. Sometimes features just slow down the classifier or may even have a negative effect on the result. Hence, *feature selection* is a technique that separates the useful from the useless features. Additionally, the more compact data set is easier to grasp and focuses "the user's attention on the most relevant variables" [WFH11, page 308]. Therefore, feature selection improves the classifier's performance and increases the clarity of the data set.

6.2 Classification of Function Pairs

SALs can consist of various functions. Hence, prioritization can help focusing on the right function pair comparisons for further inspection. Correspondingly, if a mechanism can sort and pre-classify all function pairs of the system under analysis, the user can focus on uncertain comparisons.

To achieve this, a prediction model is build (see Figure 6.1) that takes two inputs: the metrics of the function pairs under analysis plus information about these function pairs from a knowledge base. The prediction model can ask the knowledge base whether a certain function pair is already classified or not. Accordingly it will use it for training purpose or alternatively suggest a classification.

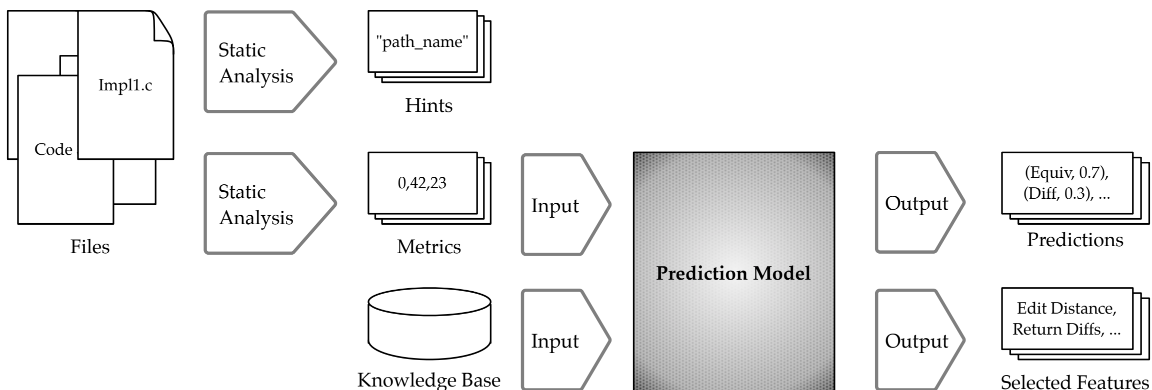

Figure 6.1: The process for the prediction model

6.3 Used Features

For a well-functioning classification algorithm, good features are essential. We identified various metrics of difference in the previous chapter (see Section 5.2). To these we added the label UserDecision. This field is the given classification contained in the knowledge base, if the function pair has been classified already. Otherwise a flag indicates that the knowledge base contains no information regarding the classification of this function pair. In this case, the prediction model will create a prediction for this instance.

Table 6.1 shows one example with the complete feature list for the POSIX and RTEMS implementations of `OS_translatePath`. The functions are listed in Appendix E.1.

Feature	Value
Name	`OS_TranslatePath`
Compared implementations	POSIX and RTEMS
UserDecision	Different
LoCDiff	26
Editing distance	1191
Code-complexity difference	7
Global function calls difference	0
Local function calls difference	0
Local global function calls difference	4
Variable usage differences	19
Project constant usage	2
Constant usage	5
Parameter writing differences	0
Global writing differences	2
Return code differences	1
Integer value differences	2
String value differences	2
Operator diffs	30
Comparison operator diffs	9
Arithmetic operator diffs	11
Logical operator diffs	5

Table 6.1: An example showing all features in use

6.4 Used Classification Algorithm

The feature vector presented above has one label (UserDecision) and 18 regular features, of which all non-labels are real numbers. Hence, as the data does not grow large quickly and performance was not a key interest in this study, we used an *instance-based* learning algorithm, where data is stored verbatim [WFH11, page 131]. An instance-based classifier categorizes an unknown example by looking at other close examples. The k-nearest-neighbors approach (k-NN) chooses the label with the highest frequency from the closest k items, for a specified k. RapidMiner uses Euclidean distance to determine how close two instances are:

$$\text{dist}(A_1, A_2) = \sqrt{\sum_{i \in [1,N]} (a_{1,i} + a_{2,i})^2}$$

where A_1 and A_2 are instances with $a_{1,i}$ being the feature i of instance A_1. As all features of the model are real numbers, we can easily compute these values.

The value which most of the nearest neighbors have as their label, is then proposed as the classification for the unknown example. Afterwards the model can calculate the confidence for a target value t of an unknown example X by taking the percentage of the k

examples that support this classification.

$$\text{confidence}_t = \frac{|\{n|\text{targetAtt}(A^n) = t \wedge A^n \in \text{kNN}(X,k)\}|}{k}$$

where $\text{kNN}(X,k)$ is a set of the k nearest neighbors of X using the distance function mentioned above.

However, if the examples that are closer to the unknown example should have a higher impact, weighted voting increases their effect onto the confidence. With weighted voting, confidence is calculated like the following:

$$\text{totalDistance} = \sum_{n \in [1,k]} \text{dist}(X, A^n)$$
$$\text{normFactor} = \max(k-1, 1)$$
$$\text{confidence}_t = \sum_{n|\text{targetAtt}(A^n)=t} \frac{1 - \frac{\text{dist}(X,A^n)}{\text{totalDistance}}}{\text{normFactor}}$$

First we calculate two constants for normalization. Afterwards, we take those instances of the k nearest neighbors that have the same target value as the value, which the classifier will predict for X. Afterwards, we sum up the confidence based on the distance of the individual instances. Hence, instances that are closer to the unknown example have a stronger impact onto the confidence.

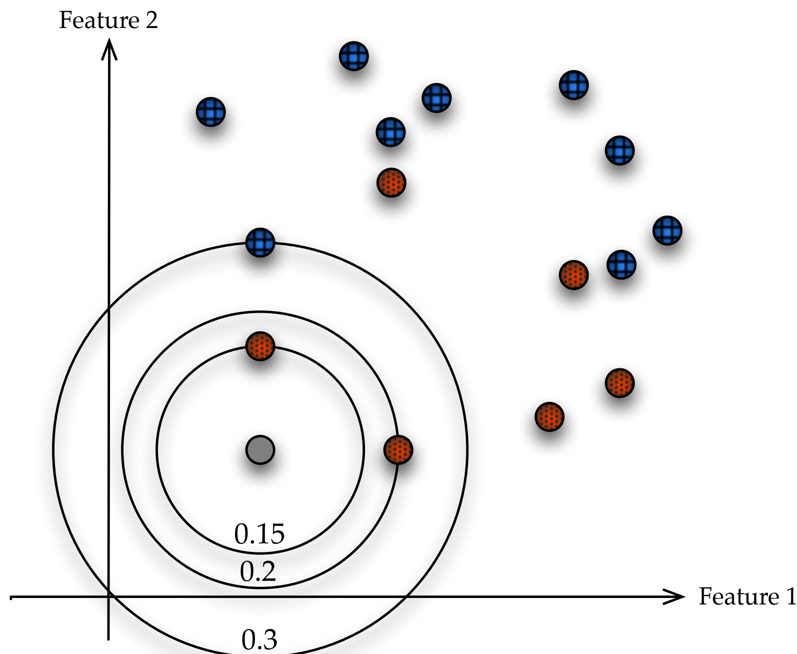

Figure 6.2: Example for k-nearest neighbors classification

Figure 6.2 shows an example of this approach. For a 2-dimensional space, we want to classify examples into red and blue instances. The red and blue dots were used to

train the model; the gray dot is an item to be classified. We marked distances from the unknown example with circles and give the exact distance at the bottom of each circle. Applying a k-NN approach with $k = 3$, we can quickly see that the model would classify the gray dot as red, because red constitutes the majority in the $k = 3$ distances. With unweighted voting, the confidence would be $\frac{2}{3}$. In weighted voting however, we calculate the confidence according to the previous formulae:

$$\text{totalDistance} = 0.15 + 0.2 + 0.3 = 0.65$$
$$\text{normFactor} = \max(2, 1) = 2$$
$$\text{confidence} = \frac{1 - \frac{0.15}{0.65}}{2} + \frac{1 - \frac{0.2}{0.65}}{2} \approx 0.73$$

We see that the blue dot's importance decreases with weighted voting, as it is further away from the unknown dot.

One problem that may occur when using k-NN models are differences between the domains of the features used: If one feature contains values between zero and one and another feature uses huge real numbers, the small numbers will dominate the model. Hence, normalization has to be performed, so that all features have an equal impact onto the model.

We applied a normalized 6-NN approach with weighted voting. Features are normalized between zero and one by calculating

$$a_i = \frac{v_i - \min v_i}{\max v_i - \min v_i}$$

where a_i represents the normalized and v_i the non-normalized value. The max and min functions are applied over the whole training set [WFH11, page 132].

6.5 Feature Selection for Filtering Hints

Including too many features into the data set can distort the performance of the classifier [WFH11, page 30]. Hence, algorithms are in use, which reduce the feature set to a more optimized one. As there are $2^{|\text{Features}|}$ possibilities for subsets of the feature set, it is usually infeasible to find the best solution, the global optimum. In contrary, heuristics try to find a local minimum with a very reduced need of computation.

Two of those approaches are forward (feature) selection and backward (feature) elimination (see Figure 6.3, the acronyms are short forms of features from Table 6.1). In forward selection, all features are added one by one. After each time a feature is added, the performance of the classifier with the new set of features is evaluated. The feature with the best performance improvement is added to the feature set. The algorithm terminates when a local optimum is reached, i.e. no added feature would improve the performance.

Backward elimination works very similar, except that the process starts with the whole feature set and removes feature by feature. Whenever a removal of a feature does not make the performance worse, the algorithm removes the feature from the feature set. Accordingly, when removal of any feature from the feature set would have a negative effect onto the performance of the classifier, the algorithm terminates.

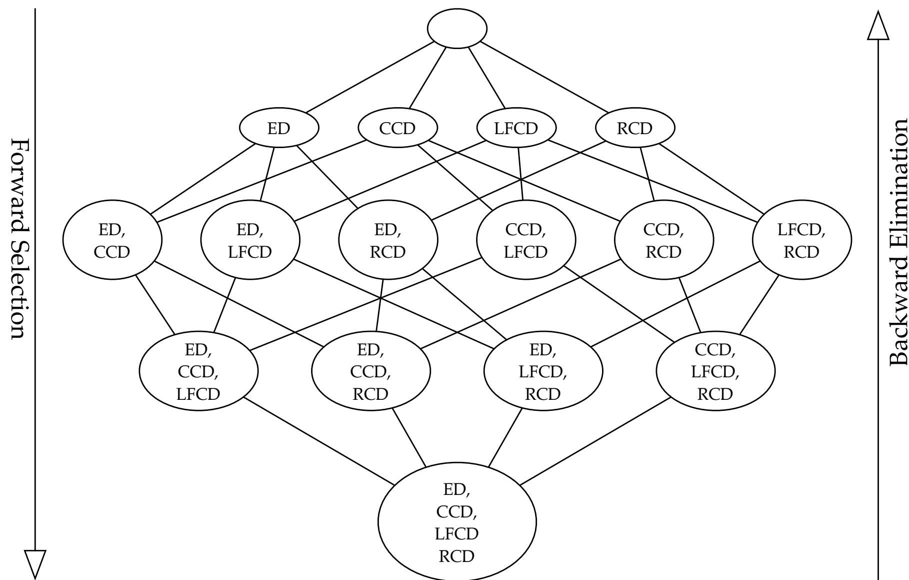

Figure 6.3: Feature selection and backward elimination, built upon [WFH11, page 311]

Both approaches find local optima, which are not necessarily the global optimum. However, they need significantly less performance evaluations, as not all possibilities are checked. In practice, these selections usually lead to good results, i.e. performance increase and significant reduction of feature set [WFH11, page 312]. Even though more sophisticated techniques exist, their application is not generally justified [WFH11, page 313].

As forward feature selection approximates the global optimum beginning with an empty set, and backward elimination starts with the complete set, forward selection tends to have fewer features in the final feature set [WFH11, page 312]. Consequently, as we want a feature set as small as possible, we applied forward selection.

However, feature selection has another big benefit besides performance. Reducing the feature set results in a more compact, more relevant data set [WFH11, page 308]. The features that remain after the selection process are the most important features for finding equivalence and differences in the code. Hence, we can take this knowledge and use it for the hints we created in the previous chapter. When taking only hints of those extractors determined relevant by the machine learning approach, we get closer to the relevant features for semantic differences. How this is used to support an analyst for equivalence analysis will be evaluated in the next chapter.

6.6 Overall Process

In order to give an overview how these features are put together, Figure 6.4 shows the whole process in the used tool, RapidMiner. In total the prediction model contains the following steps (the terms in brackets denote the RapidMiner steps which can be found in the figure):

1. The model assumes the example set represented in lines as shown in Table 6.1. The

information has to be stored in the comma-separated values (CSV) format, where each line represents an example and commas separate features.

2. The model selects the fields for classification (*Set Role*) and normalizes the data.

3. Afterwards examples are divided into training and classification set, of which the former consists of those examples that have already been classified, and hence have a valid value in the label UserDecision. The still unclassified examples form the set for classification. In RapidMiner we need to multiply the data and subsequently filter appropriately.

4. Now the training set is first transformed (the values in UsedDecision are discretized from integer numbers into the nominal values *Equivalent* or *Different*, *Discretize*) and then used for the feature selection algorithm (*Optimize Selection*). The output of the feature selection algorithm is then stored for further processing (see next chapter) in an Extensible Markup Language (XML) file (*Write Weights*).

5. The output of the feature selection algorithm (the examples from the training set with reduced features vectors) is used to train the k-NN classifier.

6. Next, the features of the example set for classification are reduced to the ones from the feature selection algorithm (*Select By Weights*). Now we can apply the just trained classifier onto this example set, returning predictions for the label UserDecision and a confidence for this prediction.

7. Lastly, the relevant data (such as the prediction and confidence) is filtered (*Select Attributes*) and stored in a CSV file (*Write CSV*).

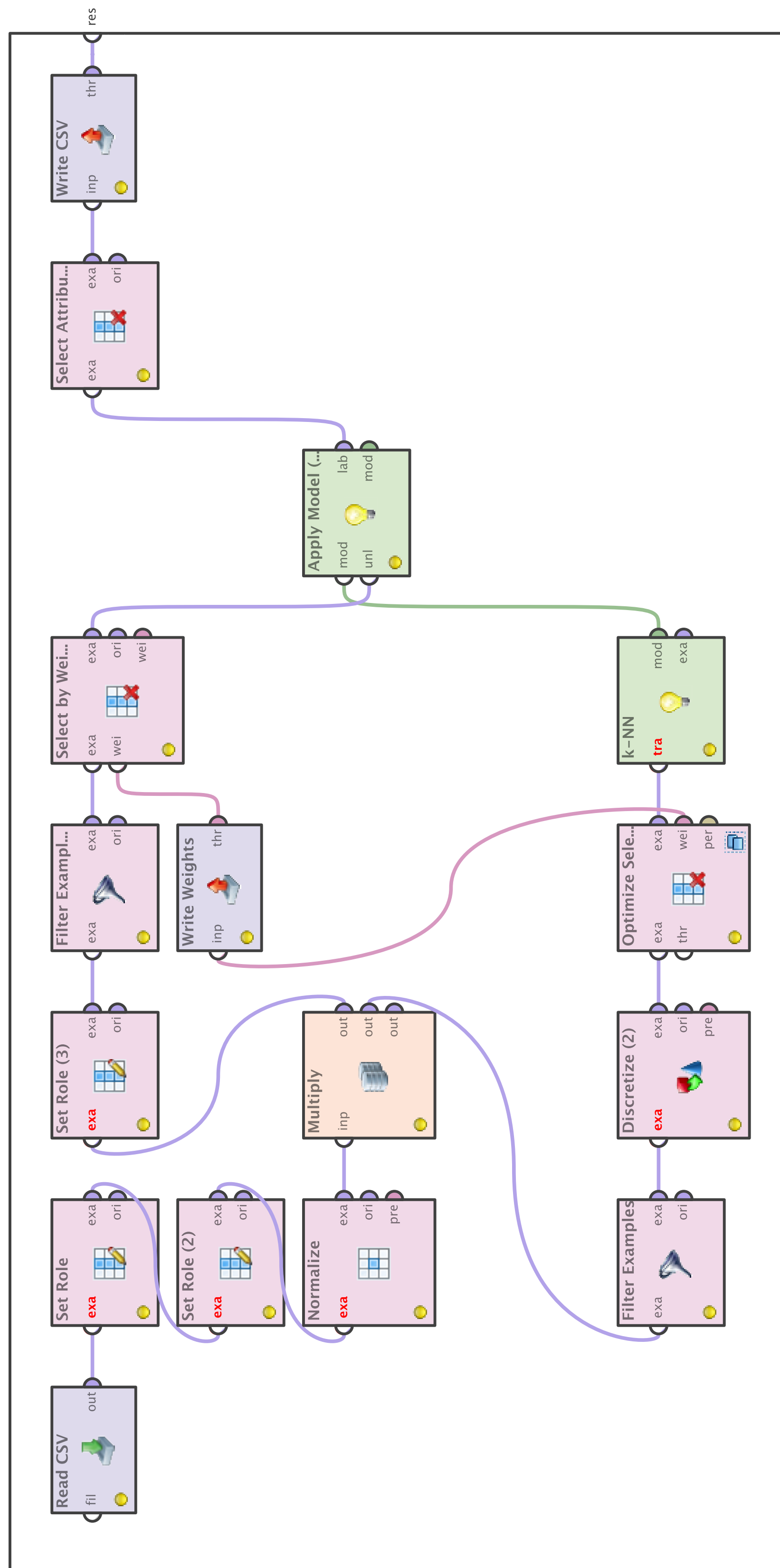

Figure 6.4: The prediction process in RapidMiner

Chapter 7

Using the User for Equivalence Analysis

This chapter describes why the user is still needed and how he can be supported in a reviewing system.

7.1 Semi-automated Equivalence Analysis

As described in Chapter 3.2.2, determining whether two functions are equivalent is undecidable. As a consequence, no technique will ever be able to determine functional equivalence in a sound and complete manner. Hence, there are three options: We can apply an approach that is incomplete, but sound, such as symbolic execution (see Chapter 4.2). This approach will not find all issues, but it can guarantee soundness for all results it produces. Second, we could apply a complete, but not necessary sound technique, such as textual differencing, where code is compared character by character. This method is complete, as it will find all possible spots of difference in code. However, such approaches often lack precision, meaning that the approach will mark all differences of which many are also part of equivalent function pairs. We will evaluate this in Chapter 8.3. Lastly, there is also a third way, the middle way. This approach is neither sound nor complete, but instead tries to combine advantages of either of the two ways: to find many semantic differences with few mistakes.

7.2 Support in a Reviewing System

When supporting an analyst during reviewing the most important aspect is to collect information and support from various sources and present only the most important of it

in a coherent fashion. The support may come from various sources:

Providing a project overview: We created a tool that gives the user a view onto the project where he can work through function pair by function pair, without searching through code bases manually. In our tool, we created a project view, where a list with all function pairs is presented to the analyst. He can sort the list by all features mentioned in the previous chapter.

Order function pairs by prediction: In addition to sorting by features we also offer sorting by the prediction created by the model described in the previous chapter. When a prediction model gives a first idea, the user can decide how much work he might want to invest. For example, if the tool is very sure that two functions are equivalent, the user might only want to quickly check and then focus on more difficult cases.

Giving hints for differences: The tool not only presents the function pairs, but also marks certain pieces of code, where it assumes relevant differences.

Hiding less relevant differences: The feature selection algorithm described in Chapter 6.5 determines which data extractors and accordingly which hints are the most relevant for the equivalence analysis as performed by the user. Hence, we only display those hints where the data extractor has an influence on the prediction model described. Thus, irrelevant differences are faded out and the (probably) relevant differences are marked in a striking color. The next chapter shows screenshots of our tool, which displays these marked hints with a strong red background and the unmarked hints in a lighter color.

Hiding hints that are repeatedly wrong: In addition to the filtering based on feature selection, we added a filtering stage where hints for differences appear various times, yet the user decided that the respective function pairs were equivalent. This means that even though there was a difference (the hint) the user decided that the difference does not affect the functional equivalence. In practice, we find all pairs of hints that appear together in equivalent function pairs. The tool assumes that if such a pair appears more than ten times, the two hints represent an equivalent aspect and are hence insignificant. However, the user can check this list of insignificant hint pairs and manually change which hints are to be displayed and which to be hidden.

Syntax highlighting: Instead of presenting the pure textual representation of the code, keywords and variables are marked with a distinct color (see the examples in the next chapter). We support the user by providing a well-known view onto the code, thereby making him feel familiar with the code. Syntax highlighting might also enhance code comprehension. The library presented in [Gru11] is used to mark the code accordingly.

Textual diffing: In the final tool we include the information of textual diffing in the background. Whenever two passages of code are textually different from each other, the background is slightly colored, so that the user attention is not drawn to these passages, but the information is available. For example, in some cases the tool does not provide any hints for differences. In order make a final judgment whether the

two functions presented are equivalent, we might want to quickly scan through the differing passages and check if the tool missed something. We can then make a decision afterwards. The diffing is performed with the library available at [Ope12b].

Line-to-line comparisons and synchronized scrolling: When the user compares two functions, the tool tries to keep the versions appropriately vertically aligned. When scrolling to a certain point in one of the implementations, the tool shows the appropriate part of the second implementation as well. This is achieved through a feature from [Ope12b].

Reports: In order to export the knowledge retrieved through the comparison of functions pairs, we added a way to export the knowledge from the tool into CSV format.

7.3 Complete Tool Process

The tool described introduces a loop into our analysis process (see Figure 7.1).

1. Static analysis creates hints and metrics of various kinds (see Chapter 5).
2. The metrics together with the existing knowledge from the knowledge base is fed into a prediction model, which creates predictions for unknown examples and a set of relevant, selected features (see Chapter 6).
3. Afterwards, the filtered static analysis results plus the predictions from the prediction model are fed into a tool and presented to an analyst.
4. The analyst looks at the tool and analyzes the code for equivalence. He makes a decision whether the two functions are equivalent or different, based upon the hints presented by the tool.
5. This decision feeds back into the knowledge base, which changes the predictions and the hints that are displayed to the analyst. Depending on the size of the system under analysis, we can start reclassifying with the new knowledge either immediately or at an appropriate point later (e.g. overnight).

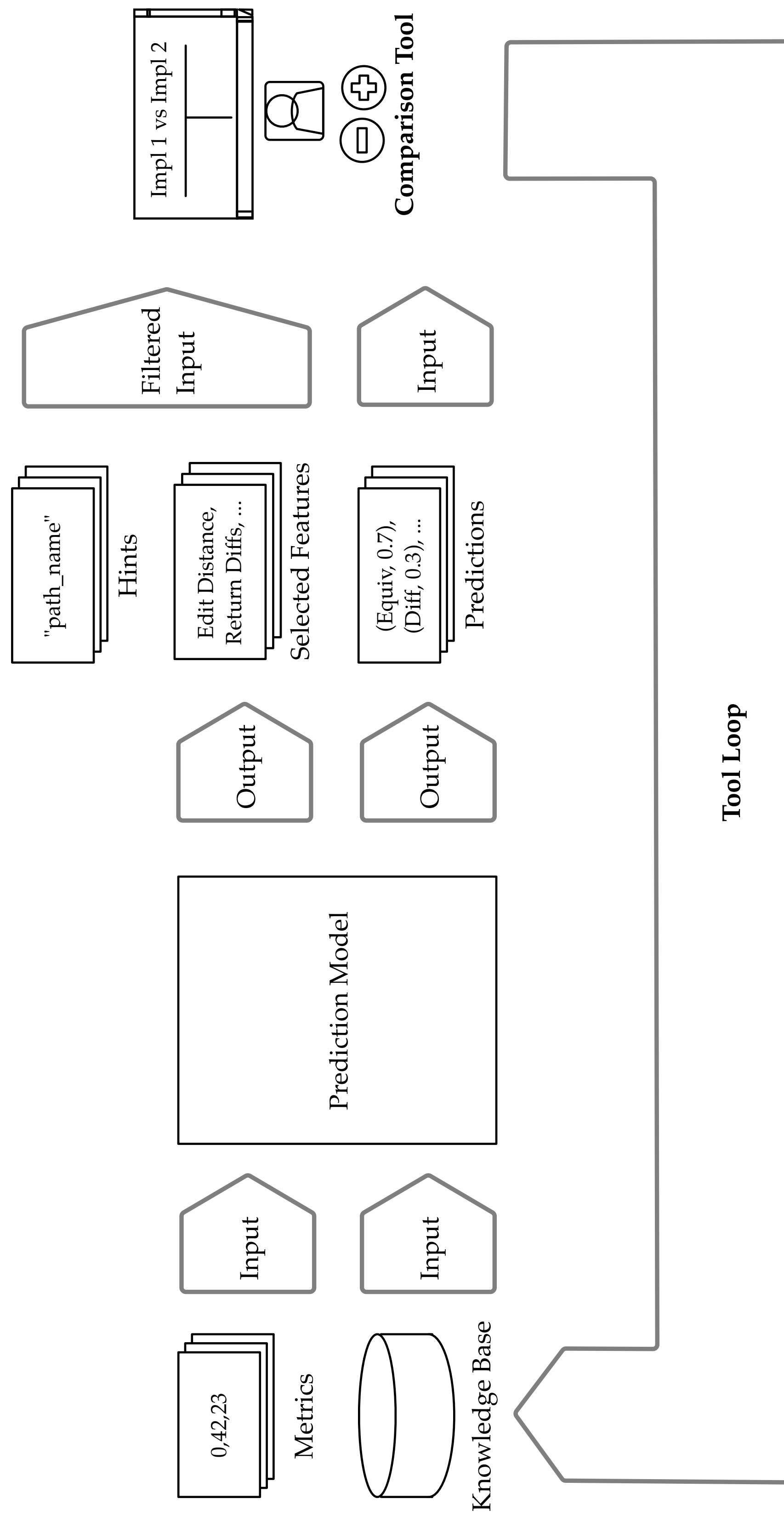

Figure 7.1: The tool loop

Part IV

Evaluation

Chapter 8

Evaluation with the Operating System Abstraction Layer

We describe the Operating System Abstraction Layer (OSAL), and analyze its code. On the way we explain the various issues found.

Goal of study: This study aims to evaluate the equivalence analysis approach presented in the previous chapters in terms of bug finding and automation potential in comparison to traditional diffing for a real industry project from the point of view of a developer of aforementioned project.

First, we will introduce the reader to the case study. Afterwards, we will analyze four research questions. Each section describes the problem, reasons about the background or shows a conducted study and answers one question, based on the experience gained. The following four questions will be evaluated:

1. Is the OSAL a representative industry project?

2. Is traditional diffing appropriate to find equivalence?

3. Is the presented approach able to find serious issues in OSAL?

4. Is the automated approach able to classify equivalence?

8.1 Introducing OSAL

The goal of this section is to give an overview over the OSAL library.

Purpose of OSAL

When a software application needs functionality of the OS such as file access, retrieving file system information or task creation, system calls offer an API to the developer. For example, when a UNIX based ANSI C program needs to know how much disk space is available on a certain disk, it can make a system call to `statvfs`, with certain parameters and retrieve the information from one of the parameters. A program running under the open source real-time operating system RTEMS [OAR12] would do the same. However, in other systems, such as the commercial real-time operating system VxWorks6 [Win12], there is no such system call. Instead, programs need to make a system call to `ioctl` with completely different parameters and a different unit for the free space. This raises a question: How can a developer write software that runs on both operating systems?

The OSAL is an abstraction layer that can be used by ANSI C code. The purpose of OSAL is to have a common interface for various system calls of different operating systems. This way, developers can share the same code base for different operating systems, such as developer's machines and the deployment system. By sharing a common code base many versioning and deployment issues can be avoided. Also continuous integration can be established more easily. OSAL is implemented for software running under UNIX, VxWorks6 and RTEMS systems and can be easily extended.

History, Applications and Resources

OSAL was initially developed by Alan Cudmore and is now maintained and extended at National Aeronautics and Space Administration (NASA) Goddard Space Flight Center in Greenbelt, MD, USA. It is openly available [NAS12d] under NASA open source agreement (NOSA)[NAS12b]. OSAL is in productive use in several space missions.

Most importantly, OSAL is part of the Core Flight System (CFS). CFS is a reusable system for flight systems, such as satellites, orbiters or space shuttles. It consists of several layers that create the abstraction needed to apply it onto several different missions (see Figure 8.1). The operating system and the board support package (BSP) enable reuse over several different hardware platforms. However, the mission's software usually also needs an interface to the hardware (such as data storage). This is possible through system calls. Unfortunately system calls are not standardized in the common operating systems that are used in space software. To reuse code on various operating systems another abstraction layer is necessary: the Operating System Abstraction Layer.

Various missions, such as the Lunar Reconnaissance Orbiter (LRO) [NAS12a] and the Solar Dynamics Observatory (SDO) [NAS12c], use CFS. LRO launched from Cape Canaveral in 2009 for collecting about 192 terabytes of data from the moon. With this data a very comprehensive map of the moon was created in order to prepare future plans to create a lunar outpost.

CFS is also part of the Solar Dynamics Observatory (SDO) project. The SDO is trying to understand the solar variations and how the magnetic fields of the sun are working and

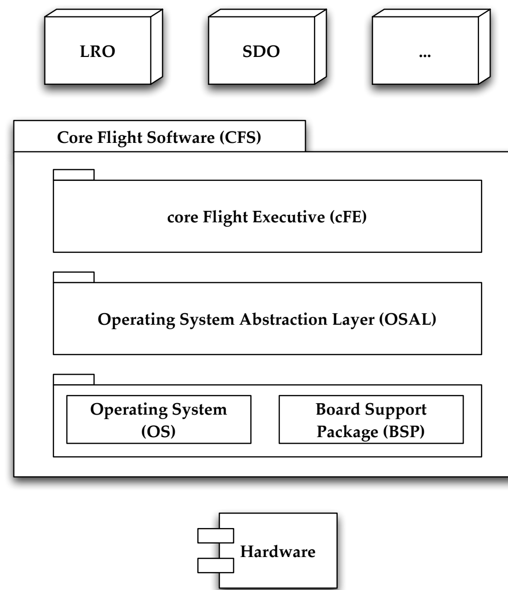

Figure 8.1: Architecture of CFS

influencing life on earth.

Architecture of OSAL

We sketched the architecture of OSAL in Figure 8.2. The OSAL API is defined in various header files. Missions and tests develop against this interface, which is then implemented by wrappers for each operating system. The OSAL can now be compiled for each operating system and placed appropriately in the running OS. The application can then call the OSAL function and the available OSAL binary will execute the call in the correct implementation accordingly.

In order to create a consistent abstraction for each implementation internal data structures are created and maintained as well. These data structures consistently store information about the OS status. For example, tables about existing file systems, message queues or semaphores are created and kept up to date for the calls coming into the OSAL.

Features of OSAL

Among others OSAL offers services for: [Yan07] [YCY11]

- Abstracted IDs and information: A generalized interface to file and object handling and identification is provided. All information is structurally identical for each OS.
- Tasks: The heart of most real-time operating systems are tasks. OSAL offers functionality to create, delete, start and delay tasks. Additional parameters like the priority of a task can be accessed and modified.
- Timer: In order to get a notification after a certain period of time, OSAL also

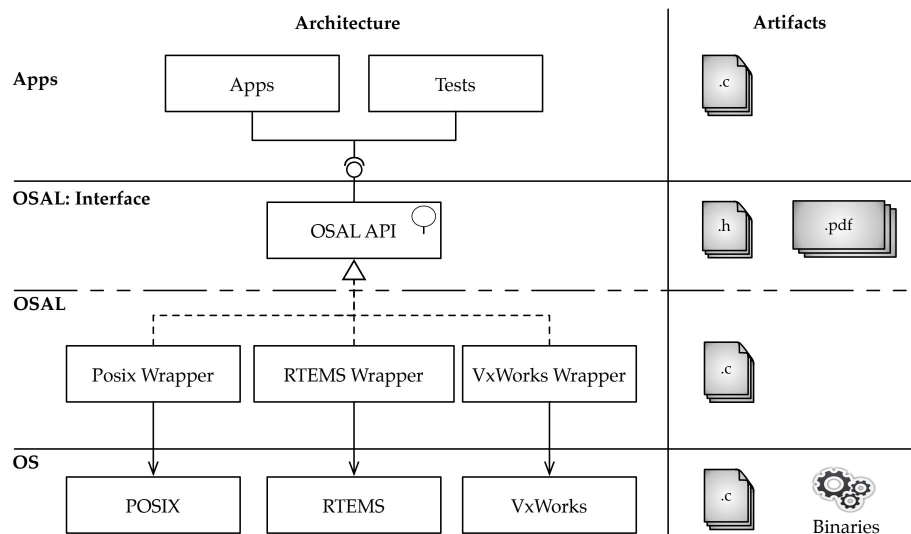

Figure 8.2: Architecture and artifacts of OSAL

implements a timer API. With these functions an application can create, delete and find timers by name or id.

- Queues: Using OSAL systems can create and delete queues, put and read bytes from the queue, as well as get some additional information.
- Semaphores: To handle passages with mutual exclusion, support for various forms of semaphores is provided.
- Files and File Systems: OSAL offers a wide range of functionality for file and file system operations. Thus a developer can manage file systems and have all kinds of access to files. File systems can be created, mounted or unmounted; files can be copied, moved, access rights can be changed and more.
- Network: Limited functionality is available for Network access: OSAL applications can retrieve information about this computer or the network this computer is part of.
- Interrupts and Exceptions: Systems can enable and disable various interrupts, as well as check for floating point unit (FPU) exceptions.

8.2 Is the OSAL a Representative Industry Project?

To derive valid consequences from this case study, we have to look at the system-under-analysis in detail first. Can we generalize results from this case study, and if so, to what extent?

8.2.1 Characteristics of OSAL

The project under analysis has applications in space software. It is used in various missions where safety is critical due to financial and personal risks.

In OSAL we analyzed 100 functions that were present in all three implementations and four functions that were only present in two implementations. This leads to $100 \times \binom{3}{2} + 4 \times \binom{2}{2} = 304$ one-to-one comparisons (or *function pairs*) in total. Of these 104 functions, 11 functions were only implemented as stubs in at least one of the three implementations. This includes functions that only return success or failure, without changing or really implementing the API. A complete list can be found in Appendix E.2.

Even though the applications of OSAL are (literally) rocket science, the functions are of short to medium length (an average of 14 executable statements per function) and low to medium complexity (average cyclomatic complexity of 5.5). It is recommended (e.g. in [WMW96]) to limit functions to a cyclomatic complexity of 10; hence, we could argue that an average complexity of six is close to a standard value in software projects. In order to give an idea how different the functions are (syntactically), we propose the average Levenshtein distance between function pairs (see Section 5.2). This metric gives a good idea of how different functions are. In OSAL the average Levenshtein distance of function pairs is 264. This indicates some relation between the implementations, but implementations are still very different.

Table 8.1 gives a short summary of the analyzed parts of the OSAL project. Tests, example apps, config-, make- and bsp-files were excluded. The top section was retrieved from [Yan07], the next four sections were generated with *understand* [Too12], and the bottom two sections were calculated within our own framework.

8.2.2 Result

The different implementations have a certain similarity and are of medium complexity. They are slightly less complex than average open source projects, e.g. given by [PSE04].

However, the OSAL is a real industry project used in extremely safety critical environments. OSAL's users need highly reliable software for real-time operating systems. Hence, the requirements to OSAL are definitely above average. As such one could argue that the results could be transferred to many less strict use cases.

A follow-up study should answer this question.

8.3 Is Traditional Diffing Appropriate to Find Equivalence?

One could image that we can find equivalences in software systems by using regular diffing. Diffing is the action of comparing two textual files, and determining the textual fragments that have to be added or removed in each version [CCD09]. The Unix command *diff* is a very prominent example, returning the minimum number of lines to be added or removed to convert one version of text into another.

But can a diffing tool really determine or at least approach such a sophisticated matter like software equivalence? To answer this question we used a state-of-the-art diffing tool that fit into our tool chain, DiffPlex [Ope12b]. We studied how many of the real equivalent functions in OSAL were syntactically (character-by-character) equal.

Metric	Value
System Under Analysis	OSAL
Analyzed Version	3.2
Developed since	August 2003
Implementations	3
Files	26
Functions	317
Lines	20065
Lines of Code	8641
Comment Lines	7513
Executable Statements	4443
Average Executable Statements per Function	14
Maximum Executable Statements	65
Sum Cyclomatic Complexity	1742
Maximum Cyclomatic Complexity	43
Average Cyclomatic Complexity	5.5
Total function pairs compared	304
Equivalent function pairs	181
Different function pairs	123
Average Levenshtein Distance	264
Avg. Levenshtein (different function pairs)	456
Avg. Levenshtein (equivalent function pairs)	133

Table 8.1: Metrics of OSAL

8.3.1 Study

First, we only split up the code files into functions. After this step 14 pairs had identical code. Second, we eliminated most aspects of different code styles, using the *uncrustify* tool (see Section 5.1). This second step led to 44 pairs of identical code. However, by manually reviewing we found out that 181 pairs of functions had identical code. Thus, even more elaborated diffing could find less than a fourth of the real equivalences (23.4%, see Figure 8.3).

One example for a function that is functionally equivalent but syntactically different can be found in Figure 8.4. We have two implementations of the OSAL function `OS_TaskExit`: an implementation for POSIX to the left, and an implementation for RTEMS to the right. We can see three differences that are irrelevant to the functional equivalence of the software abstraction layers: First, the RTEMS version declares an additional variable, `status` that is only written, but never read. Hence, neither control flow, nor external data is changed because of this difference. It has no effect to the functional equivalence described in Section 3.2.2. Second, POSIX and RTEMS have different ways to protect and free semaphores. In POSIX this is achieved by calls to `pthread_mutex_lock` and `pthread_mutex_unlock`,

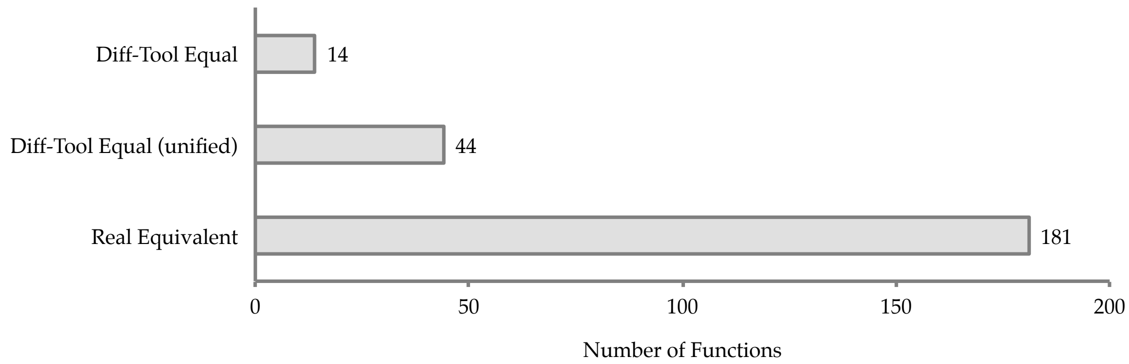

Figure 8.3: Using diffing for finding equivalence

```
..\..\..\Data\osal-3.2\src\os\posix\osapi.c
void OS_TaskExit()
{
    uint32 task_id;


    task_id = OS_TaskGetId();

    pthread_mutex_lock(&OS_task_table_mut);

    OS_task_table[task_id].free = TRUE;

    strcpy(OS_task_table[task_id].name, "");
    OS_task_table[task_id].creator            = UNINITIALIZED;
    OS_task_table[task_id].stack_size         = UNINITIALIZED;
    OS_task_table[task_id].priority           = UNINITIALIZED;
    OS_task_table[task_id].id           = UNINITIALIZED;
    OS_task_table[task_id].delete_hook_pointer  = NULL;


    pthread_mutex_unlock(&OS_task_table_mut);

    pthread_exit(NULL);

}
```

```
..\..\..\Data\osal-3.2\src\os\rtems\osapi.c
void OS_TaskExit()
{
    uint32          task_id;
    rtems_status_code status;

    task_id = OS_TaskGetId();

    status = rtems_semaphore_obtain(OS_task_table_sem, RTEMS_WAIT, RTEMS_NO_TIMEOUT);

    OS_task_table[task_id].free  = TRUE;
    OS_task_table[task_id].id= UNINITIALIZED;
    strcpy(OS_task_table[task_id].name, "");
    OS_task_table[task_id].creator            = UNINITIALIZED;
    OS_task_table[task_id].stack_size         = UNINITIALIZED;
    OS_task_table[task_id].priority           = UNINITIALIZED;

    OS_task_table[task_id].delete_hook_pointer  = NULL;
    status                  = rtems_semaphore_release(OS_task_table_sem);

    rtems_task_delete(RTEMS_SELF);


}
```

Figure 8.4: Two equivalent functions in OSAL

while RTEMS requires calls to `rtems_semaphore_obtain` and `rtems_semaphore_release`. Please note also the differing parameters used. Lastly, in the table `OS_task_table`, which stores all relevant information about existing tasks in the operating system, both implementations alter the field `id` at different locations in the code. This may lead to functional differences; however, it does not in this initialization case.

8.3.2 Result

The short study indicates that diffing can find only a very small subset of equivalent functions. Even when some more semantic information about the programming language is given, the number of equivalent functions that could be found was less than a fourth. The reason for this is that equivalence is a more subtle property than code equality. An example for these subtle textual differences can be found in Figure 8.4.

8.4 Is the Presented Approach Able to Find Serious Issues in OSAL?

In order to evaluate if the tool is of any use, we need to apply it to a real situation and see if the tool works. Can issues in the code be found? If so, how serious are they? This section shows the issues we found by applying our approach to OSAL v3.2.

8.4.1 Issues Found in OSAL

In total we found 111 issues. Please note that one function can have more than one issue. We will first give a short overview and explain the names and then go into the details afterwards.

The issues listed in Figure 8.5 are:

Return Code Differences: When two implementations use different return codes for the same situation, we call it return code differences.

Magic Numbers: Magic numbers are "numbers with special values that usually are not obvious." [FBB+99, page 126] As such, they have a negative impact on the maintainability of code [SGHS11].

Output Differences: If two programs have differences in their visual output, we call those output differences.

Global Writing Differences: We refer to global writing differences, if two programs manipulate variables accessible by all implementations in a different way.

Precondition Checking Differences: Most functions need a set of requirements to be fulfilled to properly execute their purpose. When precondition checking differences exist, these requirements are semantically differently tested.

Config Issues: Some configuration constants are not consistently used in the various implementations.

Parameter Writing Differences: Sometimes programs manipulate parameters in different ways.

Parameter Checking: This is a special case of precondition checking, where parameters are involved.

Different Check: Like with magic numbers, these are issues that relate to maintainability.

Other Issues: This category serves for issues that do not fit into one of the above-mentioned categories.

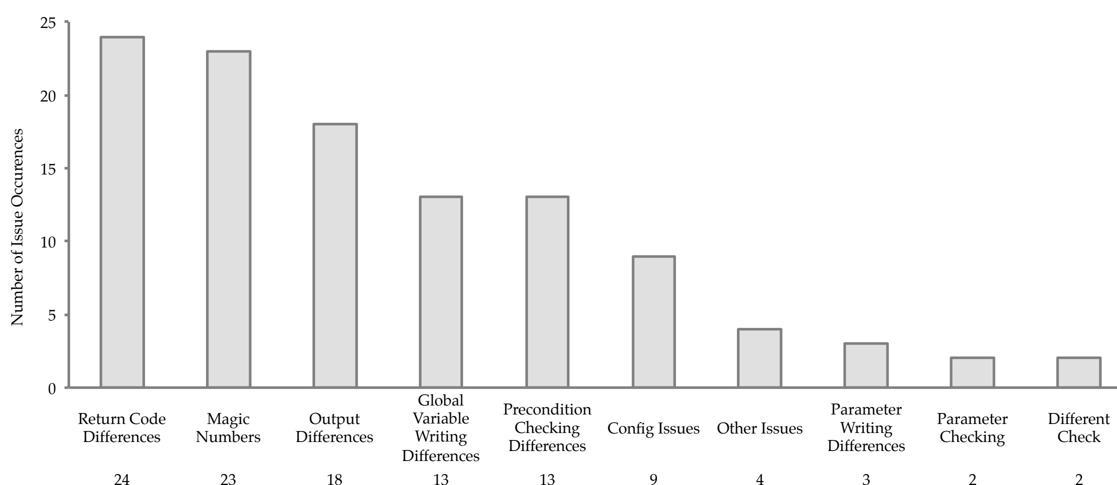

Figure 8.5: Total number of issues found in OSAL

In the following we will go into some of the bugs in detail. We classified the issues into the two possible differences identified in Chapter 3.2.2. The first part contains differences regarding the return value, the second part relates to side effects.

Return Value Issues

The first set of issues focuses on the most obvious element of functions: the return value. In an equivalent context, with equal parameters, two implementations should return the same value. We found that this is not always the case in OSAL.

Return Codes OSAL uses the return value for notifying if the function was properly executed. In this case the called OSAL function should return `OS_SUCCESS`. In every other case, an appropriate error code is defined and explained in the function's documentation.

As we knew from previous code reviews, OSAL has issues with the return codes used in the implementations. Occasionally an implementation returns unspecific error messages. `OS_BinSemCreate`, to give one example, is in charge of implementing a binary semaphore for mutual exclusion. By applying our tool, we found that in case something goes wrong during the creation of the semaphore, the POSIX implementation of OSAL

```
..\..\..\Data\osal-3.2\src\os\vxworks6\osfileapi.c
int32 OS_check_name_length(const char *path)
{
    char * name_ptr;
    char * end_of_path;
    int name_len;

    if (path == NULL)
    {
        return ( OS_FS_ERROR);
    }
```

```
..\..\..\Data\osal-3.2\src\os\rtems\osfileapi.c
int32 OS_check_name_length(const char *path)
{
    char*  name_ptr;
    char*  end_of_path;
    int name_len;

    if (path == NULL)
    {
        return ( OS_FS_ERR_INVALID_POINTER);
    }
```

Figure 8.6: An issue showing a difference in return codes in OSAL

```
if (fd == ERROR)
{
    if ( errnoGet() ==  ENOSPC )
    {
        /*
        ** There is no space to even open the file, so return 0 blocks free.
        */
        return 0;
    }
    else
    {
        return OS_FS_ERROR;
    }

}
```

Figure 8.7: An issue showing a bug of function not returning an return code in OSAL

uses the very general return code `OS_FAILURE` to indicate the lack of success. However, the implementations of VxWorks6 and RTEMS use the more specific `OS_SEM_FAILURE`, indicating that there was an issue with the semaphore. In case an application depends on either of the values, it will fail when the implementation switches.

A similar issue can be seen in Figure 8.6. In this function a wrong pointer is reported in two different ways. On the left we see the implementation of `OS_check_name_length` for VxWorks6, to the right we see the implementation for RTEMS. In both implementations a pointer is checked, but both implementations return different error codes in case of failure.

Twice implementations confused the return code with a return value. For example, `OS_fsBytesFree` (see Figure 8.7) should return the number of bytes free in a given file system. The implementation of the function should store the result in one of the pointers given as a parameter. However, in a certain error case, the VxWorks6 implementation returned a 0 in the return value instead of the parameter. This should indicate that zero bytes are free in this file system. But in this function the return value is reserved for the return code and dramatically, zero is the return code for success. So the function returns success instead of error and the parameter reserved for the result remains untouched.

The former issues were known and already corrected in the latest version, the latter issue we found during our investigations. A complete list of issues related to return codes can be found in Appendix E.3.1.

```
if (strlen(queue_name) > OS_MAX_API_NAME)        if (strlen(queue_name) >= OS_MAX_API_NAME)
{                                                 {
    return ( OS_ERR_NAME_TOO_LONG);                   return ( OS_ERR_NAME_TOO_LONG);
}                                                 }
```

Figure 8.8: An issue showing a difference in checking the string length in OSAL

Precondition and Parameter Checking Functions usually assume some constraints about the preconditions and parameters passed (see Section 3.2.2). Usually, these parameters get checked at the beginning of a function (e.g. if pointers are null). In the reviews we quite quickly realized that precondition checking in OSAL contained discrepancies. Not only differed the return codes from time to time, but also the question when or if a certain parameter is checked against forbidden values and how.

The most issues we found were related to text strings. In OSAL file and directory names are limited to a certain length that is defined in the configuration. But discrepancies in implementations exist, differing in the question if the given length maybe just as long as the value set in the configuration. One example can be seen in Figure 8.8: Before any further operation the length of a string parameter, in this case `queue_name`, is retrieved and checked against a certain constant (`OS_MAX_API_NAME`). But the implementations differ functionally: When a queue has exactly the length `OS_MAX_API_NAME`; the implementation to the right will return an error code, whereas the implementation to the left will continue with its regular procedure. A complete list of similar issues can be found in Table E.3.2.

The second issue related to parameter checking we found in the implementations of `OS_creat` and `OS_open`. These functions are used when the callee wants to create a new file or open an existing one. In both cases, a parameter is needed that describes the access level to the file. This is coded into a signed integer with predefined macros, i.e. `OS_READ_ONLY`, `OS_WRITE_ONLY`, `OS_READ_WRITE`. The different implementations then access the underlying system calls, e.g. `open()` in POSIX. But how is the parameter used? Looking at the example given in Figure 8.9 one can see that one implementation translates the parameter into a new variable, whereas the other implementation only forwards the parameter given. This is possible assuming that the codes for the OSAL function are identical with those of the underlying system call. But when looking at the issue more closely one realizes that translating the parameter contains an additional step, which is marked by the keyword `default`. This additional step is another check for validity of the given parameters.

Configuration Issues The third class of problems related to return values, is what we call configuration issues. We discovered these issues when we analyzed and compared constants used in the various implementations. Various constants, although defined in the configuration file or visible in the configuration manual, did only appear in one of the implementations (a list is given in Appendix E.5). To the user of the OSAL these settings are accessible, and they might even have a certain effect, but it differs from implementation to implementation. For this reason we suggest to hide the option from the OSAL configuration files and move it into the implementation files.

```
pthread_mutex_unlock(&OS_FDTableMutex);

switch (access)
{
case OS_READ_ONLY:
    perm = O_RDONLY;
    break;

case OS_WRITE_ONLY:
    perm = O_WRONLY;
    break;

case OS_READ_WRITE:
    perm = O_RDWR;
    break;

default:
    return OS_FS_ERROR;
}

mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH;

status =  open(local_path, perm | O_CREAT, mode);

pthread_mutex_lock(&OS_FDTableMutex);
```

```
semGive(OS_FDTableMutex);

status = creat(local_path, (int)access);

semTake(OS_FDTableMutex, WAIT_FOREVER);
```

Figure 8.9: An issue showing a difference in parameter checking in OSAL

Side Effect Differences

A second set of discrepancies between implementations focuses on side effect differences. As analyzed in Section 3.2.2, functional equivalence can be violated by either return value or side effect differences, such as manipulation of certain global variables, parameters or generally spoken, manipulation of memory that is not freed after leaving the function.

We found the following issues violating this rule.

Global Variables From checking modifications of globally defined variables (i.e. variables not defined in the parameters or in the function itself), we observed three major discrepancies. These discrepancies can be filed into three different categories.

The first issues were rotating around the `.id` field of various internal arrays. This field stores a unique identifier for semaphores, tasks, and other data structures. The implementations for VxWorks6 and RTEMS are initializing the field in the `OS_API_Init` function. In contrary, the developers of the POSIX version only set the field during the creation of new structures. Although the variable is defined locally in the .c files of each implementation, it is defined in all three implementations.

More serious are discrepancies of the `errno` variable present in two functions of OSAL, `OS_BinSemCreate` and `OS_readdir`. The `errno` variable is defined in the standard library of the C programming language and is used to store information about failure or success of a function call. As this variable is defined outside OSAL we consider it being a more serious discrepancy. A complete list can be found in Appendix E.3.5.

Third, our tool led us to the inconsistent manipulation of a variable in the VxWorks6 version of `OS_BinSemTimedWait` (an excerpt of the function is listed in Figure 8.10, VxWorks6 left, RTEMS right). This function should reserve “a binary semaphore with a timeout.” [YCY11] OSAL creates an internal data structure for each semaphore called `OS_bin_sem_table`. This structure contains a field called `current_value` storing a value indicating how many units of this resource are available. It is properly decreased at the

```
OS_bin_sem_table[sem_id].current_value--;
if (semTake(OS_bin_sem_table[sem_id].id, sys_ticks) != OK)
{
    OS_count_sem_table[sem_id].current_value++;
    return OS_SEM_TIMEOUT;
}
return OS_SUCCESS;
```

```
OS_bin_sem_table[sem_id].current_value--;

status =   rtems_semaphore_obtain(OS_bin_sem_table[sem_id].id, RTEM

switch (status)
{
case RTEMS_TIMEOUT:
    OS_bin_sem_table[sem_id].current_value++;
    status = OS_SEM_TIMEOUT;
    break;

case RTEMS_SUCCESSFUL:
    status = OS_SUCCESS;
    break;

default:
    OS_bin_sem_table[sem_id].current_value++;
    status = OS_SEM_FAILURE;
    break;
}
return status;
```

Figure 8.10: A bug from manipulating a wrong field in OSAL

```
/* Initialize Binary Semaphore Table */

for (i = 0; i < OS_MAX_BIN_SEMAPHORES; i++)
{
    OS_bin_sem_table[i].free  = TRUE;

    OS_bin_sem_table[i].creator  = UNINITIALIZED;
    strcpy(OS_bin_sem_table[i].name, "");
}

/* Initialize Counting Semaphores */
for (i = 0; i < OS_MAX_BIN_SEMAPHORES; i++)
{
    OS_count_sem_table[i].free    = TRUE;

    OS_count_sem_table[i].creator= UNINITIALIZED;
    strcpy(OS_count_sem_table[i].name, "");
}
```

```
/* Initialize Binary Semaphore Table */

for (i = 0; i < OS_MAX_BIN_SEMAPHORES; i++)
{
    OS_bin_sem_table[i].free  = TRUE;
    OS_bin_sem_table[i].id        = NULL;
    OS_bin_sem_table[i].creator  = UNINITIALIZED;
    strcpy(OS_bin_sem_table[i].name, "");
}

/* Initialize Counting Semaphore Table */
for (i = 0; i < OS_MAX_COUNT_SEMAPHORES; i++)
{
    OS_count_sem_table[i].free    = TRUE;
    OS_count_sem_table[i].id  = NULL;
    OS_count_sem_table[i].creator= UNINITIALIZED;
    strcpy(OS_count_sem_table[i].name, "");
}
```

Figure 8.11: A bug from using a wrong constant in OSAL

beginning. However, in an error case we need to revert this decrement and free the resources again, which is performed underneath. Looking at the VxWorks6 implementation more closely, we can see that it increases the counter of the `OS_count_sem_table` array, instead of the `OS_bin_sem_table` array, thereby corrupting both arrays. This was filed by the OSAL team as a bug and corrected in OSAL v3.3 [NAS12e].

Lastly, we found an issue where the initialization within `OS_API_Init` was using a wrong constant (see Figure 8.11). The implementation to the left initializes the array until the field with the number `OS_MAX_BIN_SEMAPHORES`, the implementation to the right uses the constant `OS_MAX_COUNT_SEMAPHORES`. Looking into the definition of the array `OS_count_sem_table` we can see that it is initialized with the length `OS_MAX_COUNT_SEMAPHORES` (which is also indicated by naming). Thus, when the constant is increased in the configuration file, memory outside the bounds of the array is manipulated in the implementation on the left hand side. When the constant is decreased, the array is not initialized properly.

```
sizeCopied = recvfrom(OS_queue_table[queue_id].id, da

fcntl(OS_queue_table[queue_id].id, F_SETFL, flags);

if (sizeCopied == -1 && errno == EWOULDBLOCK )
{
    *size_copied = 0;
    return OS_QUEUE_EMPTY;
}
else if (sizeCopied != size )
{
    *size_copied = 0;
    return OS_QUEUE_INVALID_SIZE;
}
```

```
if (status == RTEMS_UNSATISFIED)

{

    return OS_QUEUE_EMPTY;

}
```

Figure 8.12: A bug showing a difference in side effects in OSAL

Parameters Modification Side effects of functions may not only arise though global variables, but also when a function manipulates variables passed though call-by-reference. When a parameter is passed call-by-reference, the machine does not pass the parameter value, but the parameter's memory location. Thus a persistent manipulation of address space outside the functions local address space is possible. In ANSI C this can be achieved through the use of pointers.

Such a situation is shown in Figure 8.12. The function `OS_QueueGet` is in charge of reading a set of bytes from a queue. The function copies the bytes to a given location and posts the number of bytes that were copied into the parameter `size_copied`. Consistent manipulation of this variable is important as callees could depend on the appropriately set value. In contrast, side effects are present in the implementations for POSIX and VxWorks6, but not for RTEMS.

A list can be found in Appendix E.3.6.

Output Differences Lastly, side effects can also be manipulations of streams in and out of the program. This could be a network stream, file systems changes or printing to the system output. In total we found 18 cases of inconsistent output behavior, by checking the `printf()` calls (see Appendix E.3.8).

8.4.2 Result

We found various issues in the OSAL library. Some of the issues can be filed as less serious inconsistencies. We count the output differences and magic numbers under this category. Others, such as the modification of the `errno` variable could lead to problems in certain special cases, such as under very special use. Some issues, such as the return code issues or the manipulation of the wrong variable in `OS_BinSemTimedWait`, were already acknowledged as bugs by the OSAL team and fixed in the most recent release.

This long list of issues clearly demonstrates that our approach can find serious issues in code.

```
for (i = 0; i < OS_MAX_TASKS; i++)
{
    if (OS_task_table[i].id == pthread_id)
    {
        break;
    }
}

task_id = i;

if (task_id == OS_MAX_TASKS)
{
    return OS_ERR_INVALID_ID;
}
```

```
for (i = 0; i < OS_MAX_TASKS; i++)
{
    if (OS_task_table[i].id == vxworks_task_id)
    {
        break;
    }
}

task_id = i;

if (task_id >= OS_MAX_TASKS)
{
    return OS_ERR_INVALID_ID;
}
```

Figure 8.13: Different ways to check an array in OSAL

8.4.3 Other Benefits

Besides the issues related to functional equivalence, we spotted two more classes of issues that were pointed out by our approach, and might help to create good software.

The issues of the first class we found were so-called *magic numbers*. These are "numbers with special values that usually are not obvious." [FBB+99, page 126] Replacing magic numbers with the appropriate constants increases readability and makes changes easier [FBB+99, pages 125ff]. We found various spots in the source code where functions used integer constants instead of error codes. For example, many implementations call library functions defined by OSAL or external libraries. After the call the functions usually want to check if the operation was successful by comparing the return code against an integer value, which is usually zero. Instead [FBB+99] suggests checking if the call returned an `OS_success` or the macro for a successful operation defined by the used library. A complete table of found issues with magic numbers is shown in Appendix E.3.7.

The second issue is a matter of code style. The function `OS_TaskRegister` initializes a task within the operating system. In Figure 8.13 is an excerpt from the POSIX (left) and VxWorks6 implementation (right). At some point in the code, the implementations have to find the task in the internal data structure. Both implementations search through the array until they find an instance with a certain `id`. When this instance is found, the implementations jump out of the loop, keeping the index of the instance stored in the variable `i`. When this instance is not found, the program exits the loop at the end, the variable `i` contains the value `OS_MAX_TASKS`. Here the implementations differ: POSIX returns an error if `i` equals `OS_MAX_TASKS`, VxWorks6 also returns an error if the value is greater than `OS_MAX_TASKS`. Unless there is a further change in the code, the different implementations are functionally equal, as `i` will never be greater than `OS_MAX_TASKS`. So, if due to a code change the POSIX version jumps over a certain field, one could argue that the VxWorks6 version is safer. In any case, awareness of the difference might be very helpful when this function is changed.

These issues do not directly lead to any functional differences in code. However, various sources indicate that code conventions such as avoiding magic numbers increase maintainability (e.g. [SGHS11] or [FBB+99]) and make code less error-prone [SA04, page 34]. Even though we did not evaluate these classes of defects, this section could be seen as a teaser indicating why also non-functional equivalence might be a goal for software abstraction layers (see Section 3.2.1).

8.5 Is the Automated Approach Able to Classify Equivalence?

Now we want to analyze the classification. This section focuses on the machine learning support that we described in Chapter 6. The question is: How well can an algorithm predict if two functions are equivalent? To answer this we first need to define quality metrics to measure the 'well'. First, we need to find an appropriate competitor to compare the approach with. Second, we need to compare the performance of our approach against an appropriate competitor. Third, we want to see the results of this competition.

8.5.1 Study Design

This first section shows how prediction quality was measured, how the test and training sets were constructed and against which baseline we compared our classifier.

Measuring Prediction Quality: Precision, Recall and Accuracy

For measuring the quality of an approach, analysts very often create confusion matrices as the one shown in Table 8.2. In the confusion matrix we categorize our guesses. The first category is the question how the item is classified, i.e. positive or negative. For this thesis this is the question whether or not the classifier determines that a certain function pair is different or equal. The second category groups the pairs based upon the question if the pair is really different or equal. This is the so-called *gold standard*. This leads to four different categories of classifications. First, the function pairs that the classifier correctly marked as being different, the true positives (tp). Second, the function pairs incorrectly classified as different, the false positives (fp). Third, function pairs that were properly classified as being equivalent, the true negatives (tn) and lastly, the function pairs that the classifier guessed to be equivalent, but which really were different. These are called false negatives (fn).

With the confusion matrix we cannot calculate three important metrics: *Precision* and *recall* are common measurements in information retrieval [JHA+99]. They can be used to compare different approaches by their effectiveness related to a certain class. We use precision and recall to determine how well our approach performs on each equivalent and different function pairs individually. In contrary, *accuracy* combines both equivalent and different pairs. It is widespread in machine learning to determine the error rate of a classifier [JHA+99].

		Condition as determined by Gold standard		
		TRUE	**FALSE**	
Prediction	**Positive**	True positive (tp)	False positive (fp)	**Precision**
	Negative	False negative (fn)	True negative (tn)	
		Recall		**Accuracy**

Table 8.2: Calculation of metrics in machine learning

Precision Precision is a rate of exactness. It is a real number $\in [0,1]$ calculated by the number of elements correctly classified as *different* divided by the total number of elements classified as *different*, i.e.: (see Table 8.2)

$$Precision = \frac{tp}{tp + fp}$$

In the context of the present thesis, we calculate precision by:

$$Precision = \frac{\mid \text{DifferentPairs} \cap \text{PairsClassifiedAsDifferent} \mid}{\mid \text{PairsClassifiedAsDifferent} \mid}$$

The resulting value is a probability of how certain a user can be that a pair classified as *different* really is *different*.

Recall On the contrary, recall is a measurement for completeness. It is used to estimate the question, whether an algorithm selects all relevant elements. Thus the ratio, again a real number $\in [0,1]$, is calculated via: (see Table 8.2)

$$Recall = \frac{tp}{tp + fn}$$

or in our case:

$$Recall = \frac{\mid \text{DifferentPairs} \cap \text{PairsClassifiedAsDifferent} \mid}{\mid \text{DifferentPairs} \mid}$$

The result of this measurement is an indicator for how many of the real equivalent pairs were suggested by the machine learning approach.

Accuracy Third, accuracy (or *success rate*) determines the correctness of a classifiers prediction. It calculates the percentage of properly classified elements [JHA+99].

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn}$$

In the context of this thesis this can be calculated by:

$$Accuracy = \frac{\mid \text{CorrectlyClassifiedPairs} \mid}{\mid \text{AllPairs} \mid}$$

The accuracy is sometimes transformed into the error rate, the percentage of wrongly classified items:

$$ErrorRate = (1 - \text{accuracy})$$

	TRUE	FALSE
Positive	{A1A2, B1B2}	{D1D2}
Negative	{E1E2}	{C1C2}

Table 8.3: Calculating precision, recall and accuracy

Example This short example should clarify the calculations: Imagine we have five comparisons of functions: {A1A2, B1B2, C1C2, D1D2, E1E2}. We apply an example classification solution onto these five items and classify {A1A2, B1B2, D1D2} as different and {C1C2, E1E2} as equivalent. Through intense manual inspection we find that the real distribution (the *gold standard*) is:
Different: {A1A2, B1B2, E1E2} and equivalent: {C1C2, D1D2}.

Thus we have the following table:

We can now calculate the metrics:

$$Precision = \frac{tp}{tp + fp} = \frac{2}{3}$$

$$Recall = \frac{tp}{tp + fn} = \frac{2}{3}$$

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn} = \frac{3}{5}$$

Training and Testing

In order to get realistic predictions for performance in future we cannot calculate the performance based on classifications of the same items that were used to train the classifier. First, we need to separate training and test set [WFH11, page 148]. Obviously, the selection of those sets strongly influences the performance and hence needs to be executed with care. Second, we need to make sure that both test and training set are "representative samples of the underlying problem." [WFH11, page 149]

There are two ways to support the proper decision of splitting up data sets into training and test set: cross-validation and repetition.

Cross-validation N-fold cross-validation splits up the data set into n equally sized sets or *folds*[1]. It then uses all but one fold for training and the last fold for testing. The testing sets are rotated n times so that every sample is predicted exactly once. It is common to use two-thirds of the samples for training and one-thirds for testing ($n = 3$) [WFH11, page 152]. Of course, cross-validation can be used for many different values of n; however, for various reasons repetition of three-fold and ten-fold cross-validation are the established means to calculate performance of classifiers [WFH11, page 154].

[1]Or approximately equally if n does not divide the sample size.

Repetition The second option is repetition. In each repetition we randomly split up the items into training and test set. This is repeated *i* times. One can then average the error rates to calculate an overall error rate. This is called the *repeated holdout method for error rate estimation* [WFH11, page 153].

ZeroR classifier

Now that we know how to measure the quality of an approach, we have to find an opponent, a competitor. ZeroR is useful as a baseline, to which other classifiers can be compared.

ZeroR basically looks at the classes possible and chooses the most probable one (i.e. the class with the most samples) without looking at any data. ZeroR counts the samples of each class and afterwards always chooses the majority class. Thus ZeroR always has an accuracy of over 50%.

Example Let us have a look at the example from above (see Section 8.5.1) and see how ZeroR classification works.

In this example we had (according to the gold standard) three different and two equivalent pairs. Because different pairs are in the majority, ZeroR always guesses that function pairs are different. This leads to the confusion matrix in Table 8.4.

	TRUE	**FALSE**
Positive	{A1A2, B1B2, E1E2}	{C1C2, D1D2}
Negative	{ }	{ }

Table 8.4: Calculating precision, recall and accuracy with ZeroR

We can now calculate the same metrics for ZeroR:

$$Precision = \frac{tp}{tp + fp} = \frac{3}{5}$$

$$Recall = \frac{tp}{tp + fn} = 1$$

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn} = \frac{3}{5}$$

From this example we already see that the ZeroR classifier performs quite well, maybe better than we expected (in fact, looking at the accuracy it performed just as good as the example classifier). This can be even stronger on more biased test sets. Image a test set with 90% different pairs. The ZeroR classifier would have an accuracy of 90%, without any further knowledge about the data.

This inherent bias present in the data is why it is important to compare classifiers against naive approaches like ZeroR.

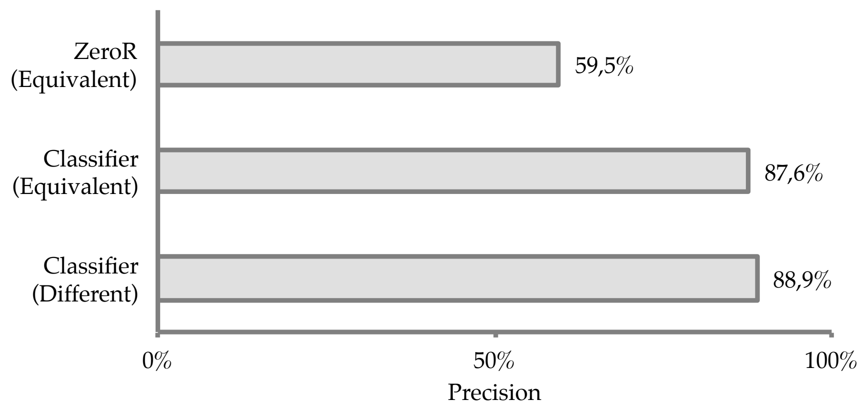

Figure 8.14: The precision of the classifier for OSAL

8.5.2 Results of the Evaluation

We used three fold cross-validation and repetition ($i = 100$) in order to evaluate the quality of the classifier. The process was run through RapidMiner [Rap11]. Data was exported into CSV files and analyzed with *R* and the built-in tools of RapidMiner.

We determined the gold standard by thorough manual inspection. In total there were 304 function pairs, of which 123 were different and 181 were equivalent (see Table 8.1). This data was used to train the classifier as described in Chapter 6.

As mentioned earlier, we can calculate precision and recall for each class (equivalent, different) of predictions. Accordingly, in the following figures, we will give the precision and recall of each class for our own classifier and for the baseline classifier ZeroR. Accuracy can be seen as a summary for both classes. Accordingly, only one value for each class is given.

Precision

Let us first have a look at the precision shown in Table 8.14. The precision gives an impression of how many of the suggested elements really are of a certain class.

First, we see here that the precision of the classifier performs around 88%. It outperforms the ZeroR precision (60%) by approximately 28 points[2]. If we would give a set of suggestions generated by the classifier to the user he would only have to correct every tenth suggestion. Second, the classifier was slightly more precise on different functions (88.9%) than on equivalent ones (87.6%). This is surprising, especially as we already found out in Section 8.3 that about one fourth of the functions are equal modulo code style. We assume that this is because of the features described in Section 6.3. Some features, like the return code analysis feature, are very strong indicators of difference if they return any hints.

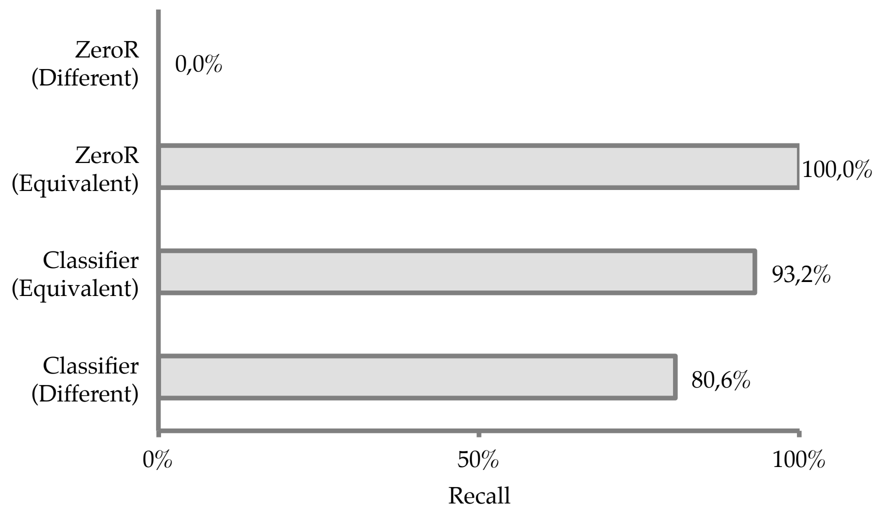

Figure 8.15: The recall of the classifier for OSAL

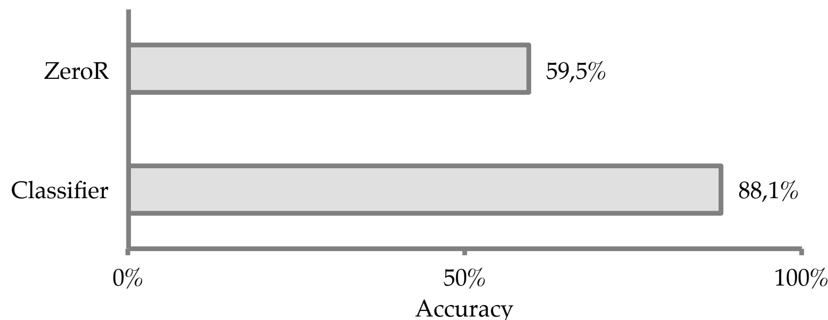

Figure 8.16: The accuracy of the classifier for OSAL

Recall

Let us now examine the recall displayed in Figure 8.15. As usually, the situation is quite the opposite of the precision. Of course, the ZeroR classifier has a recall of 100% for equivalent and 0% for different function pairs as it only suggests different pairs. Hence, it will definitely 'find' all equivalent but no different pairs. But of course, those numbers are of limited support for an empirical argument.

However, we can see that both the recall of the equivalent classifier (93%) and the recall of the different classifier (81%) are quite high. This shows that the system found 81% of the known differences in the source code and nearly all equivalent elements.

Accuracy

Finally, we want to know the accuracy, the primary indicator used in machine learning. This can be understood as a summary of the previous statistics, as it contains precision and recall of both classes into one metric. It is plotted in Figure 8.16.

[2] No precision for ZeroR(different) is given, as ZeroR does not suggest any different pairs and thus the precision cannot be calculated.

We can see in the statistic that the accuracy of the classifier (88%) surpasses the accuracy of ZeroR (59%) by far. The error rate of the classifier is at 12% whereas the error rate of ZeroR is at 41%. This shows that the classifier outperforms guessing by far and can give a relevant classification in nine out of ten cases.

Also the average variation in the repetitions is around 1.5%. This proves significance of the difference between ZeroR and the presented approach.

8.5.3 Result

The results presented above indicate that a classifier can be constructed that can give a classification with an error rate of only 12%. This is significantly better than the bias in the data given by ZeroR classification. However, the fact that there still is an error shows that reviewing is still necessary. Consequently, a tool that includes machine learning support and is still based on the user's understanding of the code seems to be the method of choice.

8.6 Threats to Validity of the OSAL Case Study

In order to understand to what extent results of this case study can be generalized, we have to look at the threats to the validity of this study and how we faced them. These threats are usually categorized into threats to internal and threats to external validity following the definition in [CS63, pages 5ff].

8.6.1 Threats to Internal Validity

Threats to internal validity doubt that the results presented actually root in the analyzed methodology or product, but rather in other facts that were not considered [CS63, pages 5ff].

The main threat to internal validity is the question of the classification as different and equivalent in the gold standard. This was created based upon known differences acknowledged by NASA, as well as differences found by the author. We opposed this threat by checking back the differences with the NASA OSAL team. However, there can always be differences in the code that remained invisible. Thus we can only base our argumentation upon the known differences. This threat was faced by thorough manual inspection. Furthermore, excessive automated testing (~150 tests) with the Microsoft Unit Testing Framework addressed threats to internal validity through bugs within the tool.

8.6.2 Threats to External Validity

Threats to external validity question the ability to generalize the results presented [CS63, page 5]. We mainly see two aspects threatening the external validity.

The first aspect is the fact that the OSAL system has a common style. It uses defined return codes and is build as a wrapper around operating system calls. The second issue is that the tool was built with OSAL as the test base. Another analysis with a second, different code base should face both of these threats.

8.7 Summary

In this chapter we evaluated the following four questions:

Is the OSAL a representative industry project? First, it is very clear that OSAL is not a toy example. The applications of OSAL should evidence this. However, the approach should be validated on a different code base to face the threat that the programming style is essential for the good results presented.

Is traditional diffing appropriate to find equivalence? Second, a small study showed that diffing is not a sufficient method to find software equivalence. It can be used to quickly eliminate a handful of comparisons; however, the vast majority needs more sophisticated solutions.

Is the presented approach able to find serious bugs in OSAL? Third, we could demonstrate by explaining plenty of differences found that the proposed solution is able to find difference issues in a safety critical, real world, industrial code base.

Is the automated approach able to classify equivalence? Fourth, by using common methods from the field of machine learning, we examined the proposed classifier and showed that the classifier can successfully classify items with 90% accuracy. However in order to classify every item appropriately, a tool, like implemented in Chapter 7, seems to be necessary.

Chapter 9

Conclusion and Future Work

In this section we describe the results of this work and analyze chances for future research in order to summarize the contributions and suggest next steps.

9.1 Wrap-Up

To sum up this work, we see five major contributions as part of this thesis:

First, we defined software equivalence analysis for software abstraction layers: Two functions m and n are functionally equivalent modulo an abstraction aspect A iff all semantic differences between m and n are side effects in the domain of A. We furthermore described a use case at NASA where absence of functional equivalence can lead to serious bugs.

Second, we classified existing and future approaches into static and dynamic methods. We explained the choice of static code analysis due to constraints in the application domain of real-time operating systems.

Third, we explored symbolic execution for equivalence analysis. We found a serious issue that was present in all implementations. Yet, symbolic execution is based upon a very exact definition of functional equivalence and furthermore assumes comparison of all systems running in one environment. It requires large amount of modeling for real world systems that is especially cumbersome in the context of SALs, as small differences in the model may directly lead to differences in behavior. These obstacles led to the conclusion that symbolic execution is not an optimal choice for determining equivalence in the context of SALs.

Fourth, we elaborated a solution combining static analysis, machine learning and user reviews. We determined differences with data extractors providing hints for discrepancies in source code, and quantified these discrepancies in metrics. Afterwards, the metrics are

used to build a prediction model that classifies function pairs and gives a confidence in how certain the model can say that a function pair is equivalent or different. The model furthermore determines the most relevant data extractors. Based on this information a tool was built that keeps only the most important hints and feeds the analyst with the information from the previous analysis. Afterwards, the analyst applies his domain knowledge and decides whether two functions are equivalent or not. This information is then fed back into the model in order to immediately improve the tool's classification.

Fifth, we applied this approach to OSAL, a library that is commonly used in space software. We demonstrated that the approach is able to find various issues in this safety critical, real world, industrial code base. Furthermore, we found that the tool could predict function pairs with an accuracy of 90%.

9.2 Outlook

Due to time constraints the scope of this thesis needed to be limited. Consequently, there are various areas for improvements and extensions. Mainly, we see two fields of future work: Improvements for the process and further evaluation in different settings.

Technically there are various ways to improve the process and correspondingly the tool developed in this thesis. Further data extractors could be created, which might increase the accuracy of the predictor and give additional hints to the analyst. For example, a deeper analysis of the CFG is one field for deeper analysis. A next step would compare access to variables by paths. Accordingly, possible paths through one function need to be determined and matched against a path in the other function, so that deviance between the two paths can be found. For the first part, symbolic execution might be of great help, for the latter part bisimulation and model checking could provide the technology needed to check different paths.

The second field for future work is evaluation of the approach. In order to further strengthen the validity of the results presented in Chapter 8, a case study on a different project could provide interesting insights. Furthermore, it would be interesting to empirically measure the improvements which we assume that using the tool provides in comparison to other forms of determining equivalence, such as symbolic execution or pure manual reviewing.

Bibliography

[ACHB11] Thanassis Avgerinos, S.K. Cha, B.L.T. Hao, and David Brumley. AEG: Automatic exploit generation. In *Memory*. NDSS, 2011.

[AEF+05] Tamarah Arons, Elad Elster, Limor Fix, S. Mador-Haim, Michael Mishaeli, Jonathan Shalev, Eli Singerman, Andreas Tiemeyer, M. Vardi, and L. Zuck. Formal verification of backward compatibility of microcode. In *Computer Aided Verification*, pages 93–134. Springer, 2005.

[AOH06] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. JDiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, December 2006.

[APV07] Saswat Anand, Corina S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to java pathfinder. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138, 2007.

[Bar09] Maureen O. Bartholome. Core Flight Software System. Technical report, NASA Goddard Space Flight Center, 2009.

[BCJ07] Fernando Berzal, J.C. Cubero, and A. Jiménez. Hierarchical program representation for program element matching. In *Proceedings of the 8th international conference on Intelligent data engineering and automated learning*, pages 467–476. Springer-Verlag, 2007.

[BD07] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering*. Prentice Hall Press, 3rd edition, 2007.

[Bin07] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, pages 104–119. IEEE Computer Society, 2007.

[BLH08] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47. ACM, 2008.

[Blu12] Jonas Blunck. CCM web page. `http://www.blunck.info/ccm.html` [Online. Last accessed: 2012-01-24], 2012.

[CCD09] Gerardo Canfora, Luigi Cerulo, and M. Di Penta. Ldiff: an Enhanced Line Differencing Tool. In *Proceedings of the 31st International Conference on Software Engineering*, pages 595–598. IEEE Computer Society, 2009.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224. USENIX Association, 2008.

[CDW04] Hao Chen, D. Dean, and David Wagner. Model checking one million lines of C code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185. Citeseer, 2004.

[CHH+06] B. Cole, D. Hakim, David Hovemeyer, R. Lazarus, W. Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 673–674. ACM, 2006.

[CKC12] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems*, 2012.

[CKP+11] Cristian Cadar, S. Khurshid, Corina S. Pasareanu, K. Sen, and Willem Visser. Symbolic Execution for Software Testing in Practice - Preliminary Assessment. *Test*, pages 1–6, 2011.

[CKY03] E. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, number May, pages 368–371. ACM, 2003.

[CS63] Donald T Campbell and Julian C Stanley. *Experimental and quasi-experimental designs for research*. Houghton Mifflin Company, Boston, 1963.

[CZB+10] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[DR90] IMM Duncan and D.J. Robson. Ordered Mutation Testing. *ACM SIGSOFT Software Engineering Notes*, 15(2):29–30, 1990.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *ACM SIGOPS Operating Systems Review*, number 5 in SOSP '01, pages 57–72. ACM Press, 2001.

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, December 2007.

[Ern03] M.D. Ernst. Static and dynamic analysis: synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27. Citeseer, 2003.

[Fag76] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, 1999.

[FH05] Xiushan Feng and A.J. Hu. Cutpoints for formal equivalence verification of embedded software. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 307–316. ACM, 2005.

[Fre12] Free Software Foundation. Gcov - Using the GNU Compiler Collection (GCC). `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html` [Online. Last accessed: 2012-01-24], 2012.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GD07] Vijay Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, pages 519–531. Springer, 2007.

[GLA+09] Dharmalingam Ganesan, Mikael Lindvall, Chris Ackermann, David McComas, and Maureen Bartholomew. Verifying architectural design rules of the flight software product line. In *Proceedings of the 13th International Software Product Line Conference*, pages 161–170. Carnegie Mellon University, 2009.

[GLR+10] Dharmalingam Ganesan, Mikael Lindvall, Lamont Ruley, Robert Wiegand, Vuong Ly, and Tina Tsui. Architectural Analysis of Systems Based on the Publisher-Subscriber Style. *2010 17th Working Conference on Reverse Engineering*, pages 173–182, October 2010.

[Gru11] Daniel Grunwald. Using AvalonEdit. Technical report, The Code Project, 2011.

[GS08] Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, July 2008.

[GS09] Benny Godlin and Ofer Strichman. Regression verification. *Proceedings of the 46th Annual Design Automation Conference*, pages 466–471, 2009.

[Hat04] Les Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, pages 1–19, 2004.

[HEJ09] Kevin J. Hoffman, Patrick Eugster, and Suresh Jagannathan. Semantics-aware trace analysis. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09*, page 453, 2009.

[HKLR11] Chris Hawblitzel, Ming Kawaguchi, S.K. Lahiri, and H. Rebelo. Mutual Summaries and Relative Termination. Technical Report c, Microsoft Research, 2011.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hor89] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. Technical Report 6, University of Wisconsin, June 1989.

[ILF09] Noha Ibrahim, F. Le Mouël, and S. Frénot. MySIM: A Spontaneous Service Integration Middleware for Pervasive Environments. In *Proceedings of the ACM International Conference on Pervasive Services (ICPS'2009)*, pages 1–10. ACM, 2009.

[JHA+99] Markus Junker, Rainer Hoch, S A P Ag, Basis Systems, and Andreas Dengel. On the evaluation of document analysis components by recall, precision, and accuracy. In *Proceedings of the Fifth International Conference on Document Analysis and Recognition, 1999. ICDAR'99.*, pages 713—-716. IEEE, 1999.

[Jos03] Andrew Josey. Conflicts between ISO/IEC 9945 (POSIX) and the Linux Standard Base. Technical report, The Open Group, 2003.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[KLR10] Ming Kawaguchi, S.K. Lahiri, and Henrique Rebelo. Conditional equivalence. Technical Report MSR-TR-2010-119. Technical report, Microsoft Research, 2010.

[KN06] Miryung Kim and David Notkin. Program Element Matching for Multi-Version Program Analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64. ACM, 2006.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004*, (c):75–86, 2004.

[Lev66] V I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[LLF+96] Jaques-Louis Lions, Lennart Lubeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Collin O'Halloran. ARIANE 5-Flight 501 Failure. Technical report, Ariane 501 Inquiry Board, 1996.

[LS92] Janusz Laski and W. Szermer. Identification of program modifications and its applications in software maintenance. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 282–290. IEEE, 1992.

[LVH10] S.K. Lahiri, K. Vaswani, and C.A.R. Hoare. Differential Static Analysis: Opportunities, Applications, and Challenges. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 201–204. ACM, 2010.

[LW94] B.H. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.

[MC04] J.I. Maletic and M.L. Collard. Supporting source code difference analysis. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM,Äô04)*, pages 210–219. IEEE, 2004.

[McC76] T.J. McCabe. A Complexity Measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[MSF05] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. An equivalence checking method for C descriptions based on symbolic simulation with textual differences. *IEICE - Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 88(12):3315, 2005.

[NAS12a] NASA. Lunar Reconnaissance Orbiter (LRO) web page. `http://lunar.gsfc.nasa.gov` [Online. Last accessed: 2012-01-24], 2012.

[NAS12b] NASA. NASA Open Source Agreement (NOSA). `http://opensource.gsfc.nasa.gov/documents/NASA_Open_Source_Agreement_1.3.pdf` [Online. Last accessed: 2012-01-24], 2012.

[NAS12c] NASA. Solar Dynamics Observatory (SDO) web page. `http://sdo.gsfc.nasa.gov/` [Online. Last accessed: 2012-01-24], 2012.

[NAS12d] NASA Goddard Space Flight Center. Operating System Abstraction Layer (OSAL) Sourcecode. `http://osal.sourceforge.net` [Online. Last accessed: 2012-01-24], 2012.

[NAS12e] NASA Goddard Space Flight Center. OSAL Version 3.3 Release Notes. Technical report, NASA, 2012.

[NNHME09] E. Nilsson-Nyman, G. Hedin, Eva Magnusson, and T. Ekman. Declarative intraprocedural flow analysis of Java source code. *Electronic Notes in Theoretical Computer Science*, 238(5):155–171, 2009.

[OAR12] OAR Corporation. Real-Time Executive for Multiprocessor Systems (RTEMS) web page. `http://www.rtems.com/` [Online. Last accessed: 2012-01-24], 2012.

[Ope12a] Open Source. Corsis web page. `http://corsis.svn.sourceforge.net/viewvc/corsis/trunk/Tenka.Text/Tenka.Text/TextMath.cs?revision=396&view=markup#l_777` [Online. Last accessed: 2012-01-24], 2012.

[Ope12b] Open Source. DiffPlex .net Diffing Library v1.2. `http://diffplex.codeplex.com/` [Online. Last accessed: 2012-01-24], 2012.

[Ope12c] Open Source. Uncrustify Source Code Beautifier web page. `http://uncrustify.sourceforge.net/` [Online. Last accessed: 2012-01-24], 2012.

[PDEP08] Suzette Person, M.B. Dwyer, S. Elbaum, and Corina S. Pasareanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.

[PJ97] A.A. Porter and P.M. Johnson. Assessing software review meetings: Results of a comparative analysis of two experimental studies. *Software Engineering, IEEE Transactions on*, 23(3):129–145, 1997.

[PSE04] J.W. Paulson, G. Succi, and A. Eberlein. An empirical study of open-source and closed-source software products. *Software Engineering, IEEE Transactions on*, 30(4):246–256, 2004.

[Rap11] Rapid-I GmbH. RapidMiner 5.0 User Manual. Technical report, Rapid-I GmbH, 2011.

[RE11] D.A. Ramos and Dawson Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *Computer Aided Verification*, pages 669–685. Springer, 2011.

[Rom02] Jan Romberg. Model-Based Deployment with Autofocus: A First Cut. In *Proceedings of the 14th Euromicro Conference on Real Time Systems (ECRTS)(Work in Progress session). IEEE Computer Society, Los Alamitos, CA*, 2002.

[SA04] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[SAB10] E.J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331. IEEE, 2010.

[SGHS11] Michael Smit, Barry Gergel, H.J. Hoover, and Eleni Stroulia. Code convention adherence in evolving software. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 504–507. IEEE, September 2011.

[SMS+07] Thanyapat Sakunkonchak, Takeshi Matsumoto, Hiroshi Saito, Satoshi Komatsu, and Masahiro Fujita. Equivalence checking in c-based system-level design by sequentializing concurrent behaviors. In *Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology*, pages 36–42. ACTA Press, 2007.

[Syn12] Synopsys. Formality - Equivalence checking for DC Ultra web page. `http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx` [Online. Last accessed: 2012-01-24], 2012.

[Too12] Scientific Toolworks. Understand - Static Code Analysis web page. `http://www.scitools.com/` [Online. Last accessed: 2012-01-24], 2012.

[Vou90] MA Vouk. On Back-to-Back Testing. *Information and Software Technology*, 32(1):34–45, 1990.

[Wal95] Stephen R. Walli. The POSIX family of standards. *StandardView*, 3(1):11–17, March 1995.

[WFH11] Ian H Witten, Eibe Frank, and Mark A Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 3rd edition, 2011.

[Win12] Wind River. Wind River VxWorks RTOS web page. `http://www.windriver.com/products/vxworks/` [Online. Last accessed: 2012-01-24], 2012.

[WMW96] A.H. Watson, T.J. McCabe, and D.R. Wallace. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. *NIST Special Publication*, 500(235):1–114, 1996.

[Yan07] Nicholas J Yanchik. Operating System Abstraction Layer (OSAL). Technical report, NASA Goddard Space Flight Center, 2007.

[YCY11] Nicholas J Yanchik, Alan Cudmore, and Ezra Yeheskeli. OS Abstraction Layer Library, v3.4. Technical report, NASA Goddard Space Flight Center, 2011.

Appendix

Mail Conversation with Alan Cudmore

Here you can find the conversation with Alan Cudmore, initiator of OSAL about the issues they were facing with POSIX.

From: Alan Cudmore <acudmore@users.sourceforge.net>
Subject: OSAL and POSIX
Date: 6. Februar 2012 16:33:29 MEZ
To: Henning <oanry@users.sourceforge.net>

```
1  Hi Henning ,
2  A little background: The OSAL comes from NASA Goddard Spaceflight
       Center , where we use it for the flight software for small
3  scientific satellites. Over 10 years ago we started looking at
       standardizing on an operating system API to allow our code to
       become
4  more portable.
5  We did start out with the idea of using the POSIX API and we
       started a research project called \"POSIX flight software\".
6
7  At the time, we were looking at Linux for desktop prototyping,
       LynxOS for real time POSIX, vxWorks ( our most commonly used
       OS ),
8  and RTEMS. We had also used Nucleus, which did not offer a POSIX
       API.
9
10 Problems we found with POSIX:
11 1. Inconsistent APIs
12 Most of our team had a great deal of experience with these
       specialized real time operating systems such as vxWorks,
       Nucleus, and
13 the even older VRTX. The APIs were clear, easy to understand, and
       provided consistent error return codes. We found POSIX to be a
14 little less intuitive, and not nearly as consistent across the
       different platforms were were interested in.
15
16 Take for example semaphores and mutexes:
```

```
17 There are pthread mutexes, system V semaphores, and posix
       semaphores. None of them quite did everything we needed, so we
18 ended up with a mixture of the calls.
19
20 2. Not all APIs are available on all OSs
21 One of the key features we use in our software are message queues.
       The POSIX message queues ( mq_receive ) is on vxWorks,
22 RTEMS, and LynxOS, but we often rely on the timeout feature in our
       software, so we needed the mq_timedreceive API. This was
23 available on RTEMS, but not the others at the time. There were a
       few other examples of these APIs not being on all platforms.
24
25 We could have made supplemental POSIX APIs or helper libraries,
       but we would have had a mixture of system includes and our
26 own re-creation of POSIX headers. It would have been confusing for
       our code reviews. Our OSAL is simple and self contained.
27
28 One area that I would re-consider is the file system API. This
       seems to be consistent across almost all of the platforms, and
       I might
29 have re-considered duplicating the standard open/close/read/write
       APIs.
30
31 But there are a couple of areas that the file API proves useful:
32 - We do have a use for path translation to reconcile paths between
       linux, RTEMS, and vxWorks, so it does help.
33 - Sometimes our code needs to force close all of the open files we
       have before we do a reset. Using the OSAL file APIs and data
34 structures, we can quickly determine which files are open and
       close them, regardless of what thread opened it.
35
36 In the end, I think we are happy with our choice over POSIX.
37
38 I hope this helps provide some of the rationale of our OSAL.
39
40 Alan
41
42
43 --
44 This message was sent to your SourceForge.net email alias via the
       web mail form.  You may reply to this message directly, or via
       https://sourceforge.net/sendmessage.php?touser=857359
45 To update your email alias preferences, please visit https://
       sourceforge.net/account
```

Example for Functional Delta

In Section 3.2.2 we defined functional delta as the functional difference between two functions. As the definition is written in logic, we can easily deduce the difference. Functions:

Function	Partition	Effect
m	strlen(path) ≥ MAX_NAME	RETURN == ERROR
	strlen(path) < MAX_NAME	RETURN == SUCCESS
m′	strlen(path) > MAX_NAME	RETURN == ERROR
	strlen(path) ≤ MAX_NAME	RETURN == SUCCESS

$$x \equiv strlen(path) \qquad A \equiv RETURN == OK$$
$$y \equiv MAX_NAME \qquad B \equiv RETURN == ERROR$$

Calculation:

$$\begin{aligned}
\Delta_{m,m'} &= \langle m \rangle \wedge \neg \langle m' \rangle = \bigvee_{(i,e) \in m_{sum}} i \wedge e \wedge \neg \bigvee_{(i,e) \in m'_{sum}} i \wedge e \\
&= ((x \geq y \wedge B) \vee (x < y \wedge A)) \wedge \neg(x > y \wedge B \vee x \leq y \wedge A) \\
&= ((x \geq y \vee x < y) \wedge (x \geq y \vee A) \wedge (B \vee x < y) \wedge (B \vee A)) \\
&\wedge ((x \leq y \vee \neg B) \wedge (x > y \vee \neg A)) \\
&= (x \geq y \vee A) \wedge (x < y \vee B) \wedge (x \leq y \vee \neg B) \wedge (x > y \vee \neg A) \\
&= (x > y \vee x = y \vee A) \wedge (x < y \vee B) \wedge (x < y \vee \neg B \vee x = y) \wedge (x > y \vee \neg A) \\
&= ((A \wedge x < y \wedge x < y) \vee (A \wedge \neg B \wedge y < y) \vee (A \wedge x < y \wedge x = y) \\
&\vee (B \wedge x > y \wedge x < y) \vee (B \wedge x > y \wedge \neg B) \vee (B \wedge x > y \wedge x = y) \\
&\vee (B \wedge x =< \wedge x < y) \vee (B \wedge x = y \wedge \neg B) \vee (B \wedge x = y \wedge x = y)) \\
&\wedge (x > y \vee \neg A) \\
&= ((A \vee x < y) \vee (A \wedge \neg B \wedge x < y) \vee (B \wedge x = y)) \wedge (x > y \vee \neg A) \\
&= (A \wedge x < y \wedge x > y) \vee (A \wedge x < y \wedge \neg A) \vee (A \wedge \neg B \wedge x < y \wedge x > y) \\
&\vee (A \wedge \neg B \wedge x < y \wedge \neg A) \vee (B \wedge x = y \wedge x > y) \vee (B \wedge x > y \wedge \neg A) \\
&= B \wedge \neg A \wedge x = y \\
&= strlen(path) = MAX_NAME \\
&\wedge RETURN == ERROR \wedge RETURN! = OK
\end{aligned}$$

Configurations

This chapter lists the configurations for the tools in use.

C.1 Uncrustify

Uncrustify is used to unify code.

```
1  #
2  # uncrustify config file for the linux kernel
3  #
4
5  indent_with_tabs  = 2      # 1=indent to level only, 2=indent with
       tabs
6  align_with_tabs       = True       # use tabs to align
7  align_on_tabstop  = True       # align on tabstops
8  input_tab_size    = 8      # original tab size
9  output_tab_size       = 8      # new tab size
10 indent_columns    = output_tab_size
11
12 indent_label      = 2      # pos: absolute col, neg: relative
     column
13
14
15 #
16 # inter-symbol newlines
17 #
18
19 nl_enum_brace     = Add # "enum {" vs "enum \n {"
20 nl_union_brace    = Add # "union {" vs "union \n {"
21 nl_struct_brace       = Add # "struct {" vs "struct \n {"
22 nl_do_brace       = Add # "do {" vs "do \n {"
23 nl_if_brace       = Add # "if () {" vs "if () \n {"
24 nl_for_brace      = Add # "for () {" vs "for () \n {"
25 nl_else_brace     = Add # "else {" vs "else \n {"
26 nl_while_brace        = Add # "while () {" vs "while () \n {"
27 nl_switch_brace   = Add # "switch () {" vs "switch () \n {"
28 nl_brace_while    = Add # "} while" vs "} \n while" - cuddle while
```

```
29 nl_brace_else      = Add # "} else" vs "} \n else" - cuddle else
30 nl_func_var_def_blk  = 1
31 nl_fcall_brace     = Add # "list_for_each() {" vs "list_for_each()\
      n{"
32 nl_fdef_brace      = Add    # "int foo() {" vs "int foo()\n{"
33 nl_after_return       = True
34 nl_before_case = 1
35 nl_func_def_args = Remove
36 nl_func_def_end = Force
37 nl_after_semicolon = Force
38 nl_end_of_file = Remove
39 nl_after_brace_open = True
40 nl_after_brace_close = True
41
42 #
43 # Source code modifications
44 #
45
46 mod_paren_on_return  = Remove # "return 1;" vs "return (1);"
47 mod_full_brace_if = Add # "if (a) a--;" vs "if (a) { a--; }"
48 mod_full_brace_for   = Add # "for () a--;" vs "for () { a--; }"
49 mod_full_brace_do = Add # "do a--; while ();" vs "do { a--; }
      while ();"
50 mod_full_brace_while = Add # "while (a) a--;" vs "while (a) { a--;
       }"
51
52 #
53 # inter-character spacing options
54 #
55
56 sp_return_paren   = Force     # "return (1);" vs "return(1);"
57 sp_sizeof_paren      = Remove # "sizeof (int)" vs "sizeof(int)"
58 sp_before_sparen  = Force     # "if (" vs "if("
59 sp_after_sparen      = Force     # "if () {" vs "if (){"
60 sp_after_cast     = Remove # "(int) a" vs "(int)a"
61 sp_inside_braces  = Add    # "{ 1 }" vs "{1}"
62 sp_inside_braces_struct = Add    # "{ 1 }" vs "{1}"
63 sp_inside_braces_enum   = Add    # "{ 1 }" vs "{1}"
64 sp_assign      = Add
65 sp_arith    = Add
66 sp_bool        = Add
67 sp_compare     = Add
68 sp_assign      = Add
69 sp_after_comma    = Add
70 sp_func_def_paren = Remove # "int foo (){" vs "int foo(){"
71 sp_func_call_paren   = Remove # "foo (" vs "foo("
72 sp_func_proto_paren  = Remove # "int foo ();" vs "int foo();"
73
74
75 #
76 # Aligning stuff
77 #
```

```
78
79 align_enum_equ_span  = 4       # '=' in enum definition
80 align_nl_cont        = True
81 align_var_def_span   = 2
82 align_var_def_inline = True
83 # align_var_def_star = True
84 align_var_def_colon  = True
85 align_assign_span = 1
86 align_struct_init_span  = 3       # align stuff in a structure init
       '= { }'
87 align_right_cmt_span = 3
88 align_pp_define_span = 8;
89 align_pp_define_gap  = 4;
90
91 cmt_star_cont        = True
92
93 indent_brace         = 0
```

Listing 3: Uncrustify options

C.2 CCM

With CCM the code complexity metric is calculated.

```
1 <ccm>
2     <analyze>
3        <folder>.</folder>
4     </analyze>
5     <recursive>no</recursive>
6     <outputXML>yes</outputXML>
7     <numMetrics>1000</numMetrics>
8 </ccm>
```

KLEE Examples

In this chapter we give examples for applying KLEE.

D.1 Side Effect Function

This example demonstrates how to check side effects in KLEE. The two functions `test1` and `test2` return the same value but have a different side effect onto the global variable.

```
#include "klee/klee.h"

int j;
int globalVariable;

static int printf(const char* format, ...){
   return j;
}

char* test1(int input){
   int n;
   n = printf("%d",input);

   if(n>2){
         return ("big\n");
   }
   else{
         return ("small\n");
   }
}

char* test2(int input){
   int n;
   n = printf("%d",input);

   globalVariable = 2;

   if(n>2){
         return ("big\n");
```

```
30     }
31     else{
32             return ("small\n");
33     }
34  }
35
36  int main(int cArgs, char* args[]){
37     int pre;
38     int sideeffect1;
39     int sideeffect2;
40     char* res1;
41     char* res2;
42
43     klee_make_symbolic(&j, sizeof(int), "j");
44     klee_make_symbolic(&globalVariable, sizeof(int), "sideEffect");
45
46     /* preserve system state */
47     pre = globalVariable;
48
49     res1 = test1(j);
50     sideeffect1 = globalVariable;
51
52     /* resume system state*/
53     globalVariable = pre;
54
55     res2 = test2(j);
56     sideeffect2 = globalVariable;
57
58     /* compare system states*/
59     klee_assert(res1 == res2);
60     klee_assert(sideeffect1 == sideeffect2);
61  }
```

Listing 4: Detecting side effects with KLEE

D.2 Establishing a Global State

In this example we demonstrate how to initialize a global state. It simulates a vehicle, changing various variables, e.g. the speed. The main function at the end of the code shows how to initiate the global state by creating various assumptions for the global variables.

```
1  // WRITTEN BY Christoph Schulze <CSCHulze@fc-md.umd.edu>
2  #include <klee/klee.h>
3  #include "stdio.h"
4  #include "stdlib.h"
5
6  #define M_NOTINIT  0
7  #define M_OFF      1
8  #define M_INIT     2
9  #define M_INACTIVE 3
10 #define M_ACTIVE   4
```

```
11
12 /* configuration variable */
13 int InitialSpeed;
14
15 double currentSpeed  = -1000.0;
16 int currentActive = 0;
17 double currentThrottleDelta = 0.0;
18
19
20 int g_dsMode = M_NOTINIT;
21 double g_dsOldDSpeed = 0.0;
22
23 int Mode(int deactivate, int activate, int onOff, int set)
24 {
25   if( g_dsMode==M_NOTINIT || !onOff )
26     g_dsMode = M_OFF;
27   else if( g_dsMode==M_OFF && onOff )
28     g_dsMode = M_INIT;
29   else if( g_dsMode==M_INIT && (set && !deactivate) )
30     g_dsMode = M_ACTIVE;
31   else if( g_dsMode==M_INACTIVE && activate )
32     g_dsMode = M_ACTIVE;
33   else if( g_dsMode==M_ACTIVE && deactivate )
34     g_dsMode = M_INACTIVE;
35
36   return (int) g_dsMode;
37 }
38
39
40 double DesiredSpeed(int mode, int accelResume, int decelSet,
       double speed)
41 {
42   double dSpeed;
43
44   if( mode==M_OFF )
45     dSpeed = 0.0;
46   else if( mode==M_INIT )
47     dSpeed = speed;
48   else if( mode==M_INACTIVE )
49     {
50       if( decelSet )
51         dSpeed = speed;
52       else
53         dSpeed = g_dsOldDSpeed;
54     }
55   else if( mode==M_ACTIVE )
56     {
57       dSpeed = g_dsOldDSpeed;
58
59       if( decelSet && !accelResume )
60         dSpeed = dSpeed - 0.01;
61       else if( accelResume && !decelSet )
```

```
62          dSpeed = dSpeed + 0.01;
63      }
64    else
65      {
66        //printf("Mode: Invalid mode: %i\n", mode);
67        exit(0);
68      }
69
70    g_dsOldDSpeed = dSpeed;
71    return dSpeed;
72  }
73
74
75  double ThrottleCtl(double dSpeed, double speed)
76  {
77    double throttleDelta;
78
79    if( speed == 0.0 )
80      throttleDelta = -0.1;
81    else
82      throttleDelta = ((dSpeed-speed)/speed) * 0.14;
83
84    if( throttleDelta < -0.1 )
85      throttleDelta = -0.1;
86    else if( throttleDelta > 0.1 )
87      throttleDelta =  0.1;
88
89    return throttleDelta;
90  }
91
92
93  void Cruise(int      onOff,
94              int      accelResume,
95              int      cancel,
96              int      decelSet,
97              int      brake,
98              int      gas,
99              double   speed,
100             int *active,
101             double *throttleDelta)
102 {
103   int activate, deactivate, mode;
104   double dSpeed;
105
106   deactivate    = cancel || brake || gas || (speed <= 25);
107   activate      = !deactivate && (accelResume != decelSet);
108
109   mode           = Mode(deactivate, activate, onOff, decelSet);
110   dSpeed         = DesiredSpeed(mode, accelResume, decelSet, speed
        );
111
112   *active        = (mode == M_ACTIVE);
```

```
113   *throttleDelta = *active ? ThrottleCtl(dSpeed, speed) : 0.0;
114   }
115
116
117 static void Plant(double currentThrottleDelta, int active, double
       inactiveThrottleDelta, double drag, double *speed)
118 {
119   if( !active ) {
120     currentThrottleDelta = inactiveThrottleDelta;
121   }
122
123   *speed = *speed + 2.0 * currentThrottleDelta + drag;
124
125   if( *speed<0 ) {
126     *speed = 0;
127   }
128 }
129
130
131
132 int main()
133 {
134     int i;
135    int  onOff;
136     int  accelResume;
137     int  cancel;
138     int  decelSet;
139     int  brake;
140     int  gas;
141     double  inactiveThrottleDelta;
142     double  drag;
143
144    klee_make_symbolic(&onOff, sizeof(onOff), "onOff");
145    klee_make_symbolic(&accelResume, sizeof(accelResume), "
          accelResume");
146    klee_make_symbolic(&cancel, sizeof(cancel), "cancel");
147    klee_make_symbolic(&decelSet, sizeof(decelSet), "decelSet");
148    klee_make_symbolic(&brake, sizeof(brake), "brake");
149    klee_make_symbolic(&gas, sizeof(gas), "gas");
150    klee_make_symbolic(&inactiveThrottleDelta, sizeof(
          inactiveThrottleDelta), "inactiveThrottleDelta");
151    klee_make_symbolic(&drag, sizeof(drag), "drag");
152    //klee_make_symbolic(&g_dsMode, sizeof(g_dsMode), "ds_mode");
153    klee_make_symbolic(&InitialSpeed, sizeof(InitialSpeed), "
          ds_mode");
154
155
156     klee_assume(g_dsMode >= 0 && g_dsMode <= 4);
157     klee_assume(onOff >= 0 && onOff <= 1);
158     klee_assume(accelResume >= 0 && accelResume <= 1);
159     klee_assume(cancel >= 0 && cancel <= 1);
160     klee_assume(decelSet >= 0 && decelSet <= 1);
```

```
161      klee_assume(brake >= 0 && brake <= 1);
162      klee_assume(gas >= 0 && gas <= 1);
163     klee_assume(inactiveThrottleDelta >= -1.0 &&
           inactiveThrottleDelta <= 1.0);
164      klee_assume(drag >= -1.0 && drag <= 1.0);
165     klee_assume(InitialSpeed >= 0 && InitialSpeed <= 100);
166
167    if (currentSpeed == -1000.0) {
168      /* currentSpeed==-1000.0 means it is not initialized yet */
169      currentSpeed = InitialSpeed;
170    }
171
172      for (i=0;i<5;i++){
173        Cruise(onOff, accelResume, cancel, decelSet, brake, gas,
              currentSpeed, &currentActive, &currentThrottleDelta);
174        Plant(currentThrottleDelta, currentActive,
              inactiveThrottleDelta, drag, &currentSpeed);
175      }
176
177     return 0;
178  }
```

Listing 5: Establishing a global state with KLEE

D.3 Checking OS_TranslatePath

Here we show how we checked `OS_TranslatePath`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <assert.h>
5
6  #include "klee/klee.h"
7
8  #include "common_types.h"
9  #include "osapi.h"
10 #include "osconfig.h"
11 #include "bsp_voltab.c"
12
13 #include "model.c"
14 #include "model.h"
15
16 #include "model_internal.c"
17
18 int main(int numArgs, char** args)
19 {
20    int i;
21    char p1[100];
22    char filename[100];
23    char mountpoint[100];
```

```
24      char localPathR[100];
25      char localPathP[100];
26      int32 returnR;
27      int32 returnP;
28
29      int32 status;
30
31      klee_make_symbolic(p1, (sizeof p1),"p1");
32      klee_make_symbolic(mountpoint, (sizeof mountpoint),"mountpoint"
           );
33      klee_make_symbolic(filename, (sizeof filename),"filename");
34      klee_make_symbolic(localPathR, (sizeof localPathR),"localPathR"
           );
35      klee_make_symbolic(localPathP, (sizeof localPathP),"localPathP"
           );
36      klee_make_symbolic(&returnR, (sizeof returnR),"returnR");
37      klee_make_symbolic(&returnP, (sizeof returnP),"returnP");
38
39      klee_assume(p1[100-1] == '\0');
40      klee_assume(mountpoint[100-1] == '\0');
41      klee_assume(filename[100-1] == '\0');
42
43      for(i = 0;i < strlen(localPathP); i++)
44      {
45         klee_assume(localPathP[i] == localPathR[i]);
46      }
47
48      assert(OS_API_Init()==0);
49      returnR = OS_TranslatePathR(filename,localPathR);
50      returnP = OS_TranslatePathP(filename,localPathP);
51
52      assert(returnR == returnP);
53      assert(OS_TranslatePathR(0,0)==OS_TranslatePathP(0,0));
54      assert(OS_TranslatePathR(p1,0)==OS_TranslatePathP(p1,0));
55      assert(OS_TranslatePathR(0,localPathR)==OS_TranslatePathP(0,
           localPathP));
56
57      return 0;
58   }
```

Listing 6: Checking OS_TranslatePath with KLEE

OSAL

This appendix shows more extended versions of some of the examples in this thesis as well as complete lists of the bugs we found.

E.1 Example Functions

In this section we will list longer versions of the examples presented in the thesis.

E.1.1 OS_TranslatePath

```
1  int32 OS_TranslatePathP(const char *VirtualPath, char *LocalPath)
2  {
3     char devname [OS_MAX_PATH_LEN];
4     char filename[OS_MAX_PATH_LEN];
5     int  NumChars;
6     int  DeviceLen;
7     int  FilenameLen;
8     int  i=0;
9
10    /*
11     ** Check to see if the path pointers are NULL
12     */
13    if (VirtualPath == NULL)
14       return OS_FS_ERR_INVALID_POINTER;
15
16    if (LocalPath == NULL)
17       return OS_FS_ERR_INVALID_POINTER;
18
19    /*
20     ** Check to see if the path is too long
21     */
22    if (strlen(VirtualPath) >= OS_MAX_PATH_LEN)
23       return OS_FS_ERR_PATH_TOO_LONG;
24
25    /*
26     ** All valid Virtual paths must start with a '/' character
```

```
27      */
28     if ( VirtualPath[0] != '/' )
29        return OS_FS_ERR_PATH_INVALID;
30
31     /*
32      ** Fill the file and device name to be sure they do not have
           garbage
33      */
34     memset((void *)devname,0,OS_MAX_PATH_LEN);
35     memset((void *)filename,0,OS_MAX_PATH_LEN);
36
37     /*
38      ** We want to find the number of chars to where the second "/"
            is.
39      ** Since we know the first one is in spot 0, we start looking
           at 1, and go until
40      ** we find it.
41      */
42     NumChars = 1;
43     while ((VirtualPath[NumChars] != '/') && (NumChars <= strlen(
          VirtualPath)))
44     {
45        NumChars++;
46     }
47
48     /*
49      ** Don't let it overflow to cause a segfault when trying to
           get the highest level
50      ** directory
51      */
52     if (NumChars > strlen(VirtualPath))
53        NumChars = strlen(VirtualPath);
54
55     /*
56      ** copy over only the part that is the device name
57      */
58     strncpy(devname, VirtualPath, NumChars);
59     devname[NumChars] = '\0'; /* Truncate it with a NULL. */
60     DeviceLen = strlen(devname);
61
62     /*
63      ** Copy everything after the devname as the path/filename
64      */
65     strncpy(filename, &(VirtualPath[NumChars]), OS_MAX_PATH_LEN);
66     FilenameLen = strlen(filename);
67
68  #if 0
69     printf("VirtualPath: %s, Length: %d\n",VirtualPath, (int)strlen
          (VirtualPath));
70     printf("NumChars: %d\n",NumChars);
71     printf("devname: %s\n",devname);
72     printf("filename: %s\n",filename);
```

```
73 #endif
74
75     /*
76      ** look for the dev name we found in the VolumeTable
77      */
78     for (i = 0; i < NUM_TABLE_ENTRIES; i++)
79     {
80        if (OS_VolumeTable[i].FreeFlag == FALSE &&
81              strncmp(OS_VolumeTable[i].MountPoint, devname,NumChars
                 ) == 0)
82        {
83           break;
84        }
85     }
86
87     /*
88      ** Make sure we found a valid drive
89      */
90     if (i >= NUM_TABLE_ENTRIES)
91     {
92        //assert(0);
93        return OS_FS_ERR_DRIVE_NOT_CREATED;
94     }
95
96     /*
97      ** copy over the physical first part of the drive
98      */
99     strncpy(LocalPath,OS_VolumeTable[i].PhysDevName,
          OS_MAX_LOCAL_PATH_LEN);
100    NumChars = strlen(LocalPath);
101
102    /*
103     ** Add the device name ( Linux, Cygwin, OS X only )
104     */
105    strncat(LocalPath, OS_VolumeTable[i].DeviceName, (
          OS_MAX_LOCAL_PATH_LEN - NumChars));
106    NumChars = strlen(LocalPath);
107
108    /*
109     ** Add the file name
110     */
111    strncat(LocalPath, filename, (OS_MAX_LOCAL_PATH_LEN - NumChars)
          );
112
113 #if 0
114    printf("Result of TranslatePath = %s\n",LocalPath);
115 #endif
116
117    return OS_FS_SUCCESS;
118
119 } /* end OS_TranslatePath */
120
```

```
121 int32 OS_check_name_lengthP(const char *path)
122 {
123     char* name_ptr;
124     char* end_of_path;
125     int name_len;
126
127     if (path == NULL)
128     {
129         return OS_FS_ERR_INVALID_POINTER;
130     }
131
132
133     if (strlen(path) > OS_MAX_PATH_LEN)
134     {
135         return OS_FS_ERROR;
136     }
137
138     /* checks to see if there is a '/' somewhere in the path */
139     name_ptr = strrchr(path, '/');
140     if (name_ptr == NULL)
141     {
142         return OS_FS_ERROR;
143     }
144
145     /* strrchr returns a pointer to the last '/' char, so we
            advance one char */
146     name_ptr = name_ptr + sizeof(char);
147
148     /* end_of_path points to the null terminator at the end of the
             path */
149     end_of_path = strrchr(path,'\0');
150
151     /* pointer subraction to see how many characters there are in
            the name */
152     name_len = ((int) end_of_path - (int)name_ptr) / sizeof(char);
153
154     if( name_len > OS_MAX_FILE_NAME)
155     {
156         return OS_FS_ERROR;
157     }
158
159     return OS_FS_SUCCESS;
160
161 }/* end OS_check_name_length */
```

Listing 7: OS_TranslatePath for POSIX

```
1 /*rtems*/
2 int32 OS_TranslatePathR(const char *VirtualPath, char *LocalPath)
3 {
4    /*
5     ** Check to see if the path pointers are NULL
```

```
6        */
7       if (VirtualPath == NULL)
8       {
9          //assert(0);
10         return OS_FS_ERR_INVALID_POINTER;
11      }
12
13      if (LocalPath == NULL)
14      {
15         return OS_FS_ERR_INVALID_POINTER;
16      }
17
18      /*
19       ** Check to see if the path is too long
20       */
21      if (strlen(VirtualPath) >= OS_MAX_PATH_LEN)
22      {
23         return OS_FS_ERR_PATH_TOO_LONG;
24      }
25
26      /*
27       ** All valid Virtual paths must start with a '/' character
28       */
29      if ( VirtualPath[0] != '/' )
30      {
31         return OS_FS_ERR_PATH_INVALID;
32      }
33
34      /*
35       ** In the RTEMS version of the OSAL, the virtual paths are the
            same
36       ** as the physical paths. So translating a path is simply
           copying it over.
37       */
38      strncpy(LocalPath, VirtualPath, strlen(VirtualPath));
39      LocalPath[strlen(VirtualPath)] = '\0'; /* Truncate it with a
          NULL. */
40      /*
41       #ifdef DEBUG
42          printf("VirtualPath: %s, Length: %d\n",VirtualPath, (int)
               strlen(VirtualPath));
43          printf("LocalPath: %s, Length: %d\n",LocalPath, (int)strlen
               (LocalPath));
44       #endif
45       */
46      return OS_FS_SUCCESS;
47
48  } /* end OS_TranslatePath */
```

Listing 8: OS_TranslatePath for RTEMS

E.2 List of Analyzed Functions

We give a list of all analyzed functions. A ✓ means that a function is analyzed, a ∅ means that no function was available and a ⊙ means that only a mock implementations exists. A mock implementation is a function that does not intend to really implement the function, e.g. it always returns `SUCCESS` without following the specification.

Method	Posix	VxWorks6	RTEMS
OS_API_Init	✓	✓	✓
OS_BinSemCreate	✓	✓	✓
OS_BinSemDelete	✓	✓	✓
OS_BinSemFlush	✓	✓	✓
OS_BinSemGetIdByName	✓	✓	✓
OS_BinSemGetInfo	✓	✓	✓
OS_BinSemGive	✓	✓	✓
OS_BinSemTake	✓	✓	✓
OS_BinSemTimedWait	✓	✓	✓
OS_check_name_length	✓	✓	✓
OS_chkfs	⊙	✓	⊙
OS_chmod	✓	✓	✓
OS_close	✓	✓	✓
OS_closedir	✓	✓	✓
OS_CompAbsDelayedTime	✓	∅	✓
OS_CountSemCreate	✓	✓	✓
OS_CountSemDelete	✓	✓	✓
OS_CountSemGetIdByName	✓	✓	✓
OS_CountSemGetInfo	✓	✓	✓
OS_CountSemGive	✓	✓	✓
OS_CountSemTake	✓	✓	✓
OS_CountSemTimedWait	✓	✓	✓
OS_cp	✓	✓	✓
OS_creat	✓	✓	✓
OS_FDGetInfo	✓	✓	✓
OS_FileOpenCheck	✓	✓	✓
OS_FindCreator	✓	✓	✓
OS_FPUExcGetMask	⊙	✓	⊙
OS_FPUExcSetMask	⊙	✓	⊙
OS_FS_GetErrorName	✓	∅	✓
OS_FS_GetPhysDriveName	✓	✓	✓
OS_FS_Init	✓	✓	✓
OS_fsBlocksFree	✓	✓	✓
OS_fsBytesFree	✓	✓	✓
OS_GetErrorName	✓	✓	✓
OS_GetLocalTime	✓	✓	✓
OS_HeapGetInfo	⊙	✓	✓
OS_initfs	✓	✓	✓
OS_IntAttachHandler	⊙	✓	✓
OS_IntDisable	⊙	✓	✓
OS_IntEnable	⊙	✓	✓
OS_IntLock	⊙	✓	✓
OS_IntUnlock	⊙	✓	✓
OS_lseek	✓	✓	✓
OS_Milli2Ticks	✓	✓	✓
OS_mkdir	✓	✓	✓
OS_mkfs_posix	✓	✓	✓
OS_ModuleInfo	✓	✓	✓
OS_ModuleLoad	✓	✓	✓
OS_ModuleTableInit	✓	✓	✓
OS_ModuleUnload	✓	✓	✓
OS_mount	✓	✓	✓
OS_MutSemCreate	✓	✓	✓
OS_MutSemDelete	✓	✓	✓
OS_MutSemGetIdByName	✓	✓	✓
OS_MutSemGetInfo	✓	✓	✓
OS_MutSemGive	✓	✓	✓
OS_MutSemTake	✓	✓	✓
OS_mv	✓	✓	✓
OS_NetworkGetHostName	✓	✓	✓
OS_NetworkGetID	✓	✓	⊙
OS_open	✓	✓	✓
OS_opendir	✓	✓	✓
OS_printf	✓	✓	✓
OS_QueueCreate	✓	✓	✓
OS_QueueDelete	✓	✓	✓
OS_QueueGet	✓	✓	✓
OS_QueueGetIdByName	✓	✓	✓
OS_QueueGetInfo	✓	✓	✓
OS_QueuePut	✓	✓	✓
OS_read	✓	✓	✓
OS_readdir	✓	✓	✓
OS_remove	✓	✓	✓
OS_rename	✓	✓	✓
OS_rmdir	✓	✓	✓
OS_rmfs	✓	✓	✓
OS_SetLocalTime	✓	✓	✓
OS_ShellOutputToFile	✓	✓	✓
OS_stat	✓	✓	✓
OS_SymbolLookup	✓	✓	✓
OS_SymbolTableDump	⊙	⊙	⊙
OS_TaskCreate	✓	✓	✓
OS_TaskDelay	✓	✓	✓
OS_TaskDelete	✓	✓	✓
OS_TaskExit	✓	✓	✓
OS_TaskGetId	✓	✓	✓
OS_TaskGetIdByName	✓	✓	✓
OS_TaskGetInfo	✓	✓	✓
OS_TaskInstallDeleteHandler	✓	✓	✓
OS_TaskRegister	✓	✓	✓
OS_TaskSetPriority	✓	✓	✓
OS_Tick2Micros	✓	✓	✓
OS_TimerAPIInit	✓	✓	✓
OS_TimerCreate	✓	✓	✓
OS_TimerDelete	✓	✓	✓
OS_TimerGetIdByName	✓	✓	✓
OS_TimerGetInfo	✓	✓	✓
OS_TimerSet	✓	✓	✓
OS_TimerSignalHandler	✓	✓	✓
OS_TimespecToUsec	✓	✓	∅
OS_TranslatePath	✓	✓	✓
OS_unmount	✓	✓	✓
OS_UsecToTimespec	✓	✓	✓
OS_write	✓	✓	✓

Table E.1: All analyzed functions of OSAL

E.3 Issues in OSAL

This section lists all issues we found in OSAL. For details, please refer to Section 8.4.1.

E.3.1 Return Code Issues

These functions have discrepancies in their return codes. The table lists those functions where differences exist, and names the return codes that appear in the respective implementations.

Method	Posix	VxWorks6	RTEMS
OS_fsBlocksFree		0	
OS_fsBytesFree		0	
OS_BinSemCreate	OS_ERROR	OS_SEM_FAILURE	OS_SEM_FAILURE
OS_BinSemTimedWait	OS_SEM_FAILURE		OS_SEM_FAILURE
OS_check_name_length	OS_FS_ERR_INVALID_POINTER		OS_FS_ERR_INVALID_POINTER
OS_chkfs	OS_FS_UNIMPLEMENTED	OS_FS_ERR_INVALID_POINTER OS_FS_ERR_PATH_TOO_LONG OS_FS_ERROR OS_FS_SUCCESS	OS_FS_ERROR
OS_CountSemCreate	OS_ERROR	OS_SEM_FAILURE	OS_SEM_FAILURE
OS_CountSemTimedWait	OS_SEM_FAILURE		OS_SEM_FAILURE
OS_cp	OS_FS_ERROR	OS_FS_ERROR	
OS_fsBlocksFree		OS_FS_ERR_PATH_TOO_LONG	
OS_initfs	OS_FS_ERR_PATH_TOO_LONG	OS_FS_ERR_INVALID_POINTER OS_FS_ERR_DRIVE_NOT_CREATED OS_FS_ERR_PATH_TOO_LONG	OS_FS_ERR_INVALID_POINTER OS_FS_ERR_DRIVE_NOT_CREATED
OS_IntAttachHandler		OS_INVALID_POINTER OS_ERROR	OS_INVALID_INT_NUM OS_INVALID_POINTER OS_ERROR
OS_mkfs		OS_FS_ERR_INVALID_POINTER OS_FS_ERR_DRIVE_NOT_CREATED	OS_FS_ERR_INVALID_POINTER OS_FS_ERR_DRIVE_NOT_CREATED
OS_mount			OS_FS_ERR_INVALID_POINTER
OS_MutSemCreate	OS_ERROR	OS_SEM_FAILURE	OS_SEM_FAILURE
OS_QueueGet	OS_ERROR	OS_ERROR	
OS_QueuePut		OS_ERROR	OS_ERROR
OS_SymbolTableDump	OS_ERR_NOT_IMPLEMENTED	OS_ERROR	OS_ERR_NOT_IMPLEMENTED
OS_TimerCreate	OS_TIMER_ERR_UNAVAILABLE		OS_TIMER_ERR_UNAVAILABLE
OS_TranslatePath	OS_FS_ERR_DRIVE_NOT_CREATED	OS_FS_ERR_DRIVE_NOT_CREATED	
OS_HeapGetInfo		OS_SUCCESS	OS_SUCCESS
OS_IntDisable		OS_ERROR	
OS_IntEnable		OS_ERROR	
OS_IntLock	OS_SUCCESS	intLock()	(int32)rtems_int_level

Table E.2: Summary of return code issues in OSAL

E.3.2 Precondition Checking Issues

This table lists functions that differ in their preconditions.

Method	Variable Name	Posix	VxWorks6	RTEMS
OS_CountSemCreate	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_CountSemGetIdByName	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_BinSemCreate	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_BinSemGetIdByName	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_MutSemCreate	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_MutSemGetIdByName	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_QueueCreate	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_QueueGetIdByName	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_TaskCreate	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_TaskGetIdByName	OS_MAX_API_NAME	$>$	$\geq$	$\geq$
OS_fsBlocksFree	OS_MAX_PATH_LEN		$\geq$	
OS_initfs	devname, volname		$\geq$	
OS_initfs	devname, volname		== null	== null
OS_mount	devname, mountpoint			== null

Table E.3: Summary of issues with precondition checking in OSAL

E.3.3 Parameter Checking Issues

These functions check the parameters in a differing way.

Method	Posix	VxWorks6	RTEMS
OS_creat	OS_READ_ONLY OS_WRITE_ONLY OS_READ_WRITE		OS_READ_ONLY OS_WRITE_ONLY OS_READ_WRITE
OS_open	OS_READ_ONLY OS_WRITE_ONLY OS_READ_WRITE		OS_READ_ONLY OS_WRITE_ONLY OS_READ_WRITE

Table E.4: Summary of issues with parameter checking in OSAL

E.3.4 Configuration Issues

These functions contain OSAL constants that are only used within this function. Maybe one should relocate the constants into the implementation.

Method	Posix	VxWorks6	RTEMS
OS_API_Init		OS_UTILITY_TASK_ON OS_UTI*TASK_PRIORITY OS_UTI*TASK_STACK_SIZE OS_BUFFER_MSG_DEPTH OS_BUFFER_SIZE	
OS_TaskCreate		OS_FP_ENABLED	OS_FP_ENABLED
OS_printf		OS_UTILITY_TASK_ON	
OS_ModuleLoad			OS_STATIC_LOADER
OS_ModuleUnload			OS_STATIC_LOADER
OS_ModuleInfo			OS_STATIC_LOADER
OS_NetworkGetID		OS_INCLUDE_NETWORK	
OS_NetworkGetHostName		OS_INCLUDE_NETWORK	
OS_ModuleTableInit			OS_STATIC_LOADER

Table E.5: Summary of unique constants within functions in OSAL

E.3.5 Writing of Global Variables Issues

These functions differ in the way they manipulate variables outside the function.

Method	Posix	VxWorks6	RTEMS
OS_API_Init	OS_mut_sem_table[#].nested_value	OS_task_table[#].id OS_bin_sem_table[#].id OS_count_sem_table[#].id OS_mut_sem_table[#].id	OS_task_table[#].id OS_bin_sem_table[#].id OS_count_sem_table[#].id OS_mut_sem_table[#].id
OS_BinSemCreate	errno OS_bin_sem_table[#].id		
OS_BinSemDelete		OS_bin_sem_table[#].id	OS_bin_sem_table[#].id
OS_BinSemTimedWait		OS_count_sem_table[#].current_value	
OS_CountSemCreate	OS_count_sem_table[#].id		
OS_CountSemDelete		OS_count_sem_table[#].id	OS_count_sem_table[#].id
OS_MutSemCreate		OS_mut_sem_table[#].id	
OS_MutSemDelete		OS_mut_sem_table[#].id	OS_mut_sem_table[#].id
OS_printf	msg_buffer[#]		msg_buffer[#]
OS_QueueCreate	OS_queue_table[#].id		
OS_readdir		errno	
OS_TaskCreate		OS_task_table[#].id	
OS_TaskSetPriority	OS_task_table[#].priority		

Table E.6: Summary of issues with global variable writing in OSAL

E.3.6 Parameter Writing Issues

These functions differ in the way they manipulate the parameters.

Method	Posix	VxWorks6	RTEMS
OS_QueueGet[1]	*size_copied	*size_copied	
OS_HeapGetInfo		*heap_prop.free_bytes	*heap_prop.free_bytes
		*heap_prop.free_blocks	*heap_prop.free_blocks
		*heap_prop.largest_free_block	*heap_prop.largest_free_block
OS_FPUExcGetMask		*mask	

Table E.7: Summary of issues with parameter manipulation in OSAL

E.3.7 Magic Number Issues

These functions contain values that should be replaced with constants, where possible.

E.3.8 Output Difference Issues

These functions contain differences in their output behavior.

[1]The value of size_copied is not reset to 0 in the error case in RTEMS.

<table>
<tr><th>Method</th><th>Posix</th><th>VxWorks6</th><th>RTEMS</th><th>Comment</th></tr>
<tr><td>OS_BinSemCreate</td><td>666</td><td></td><td></td><td>S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH</td></tr>
<tr><td>OS_CountSemCreate</td><td>666</td><td></td><td></td><td>S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH</td></tr>
<tr><td>OS_BinSemTimedWait</td><td>0
100
1000</td><td></td><td></td><td></td></tr>
<tr><td>OS_CountSemTimedWait</td><td>0
100
1000</td><td></td><td></td><td></td></tr>
<tr><td>OS_chkfs</td><td></td><td>0
1</td><td></td><td>Use error codes.</td></tr>
<tr><td>OS_cp</td><td></td><td>1</td><td></td><td></td></tr>
<tr><td>OS_HeapGetInfo</td><td></td><td></td><td>0</td><td>Use error codes.</td></tr>
<tr><td>OS_initfs</td><td>32
30</td><td>100
32
30</td><td></td><td></td></tr>
<tr><td>OS_mkfs</td><td></td><td>32
100
1</td><td></td><td></td></tr>
<tr><td>OS_ModuleUnload</td><td></td><td></td><td>0</td><td>Use error codes.</td></tr>
<tr><td>OS_MutSemDelete</td><td>−1
0</td><td></td><td></td><td>Why init? Use error codes.</td></tr>
<tr><td>OS_QueueCreate</td><td>1</td><td></td><td></td><td>Use error codes.</td></tr>
<tr><td>OS_QueueGet</td><td>1
1000
0</td><td>0</td><td></td><td>Use error codes.</td></tr>
<tr><td>OS_ShellOutputToFile</td><td>777</td><td></td><td></td><td>Use constants.</td></tr>
<tr><td>OS_TimerDelete</td><td>status< 0</td><td>status< 0</td><td>status!=RTEMS_SUCCESSFUL</td><td>Use error codes.</td></tr>
<tr><td>OS_TimerSet</td><td>status< 0</td><td>status< 0</td><td>status!=RTEMS_SUCCESSFUL</td><td>Use error codes.</td></tr>
<tr><td>OS_FindCreator</td><td>0</td><td></td><td></td><td>Use error codes.</td></tr>
<tr><td>OS_FS_Init</td><td>0</td><td></td><td></td><td>Use error codes.</td></tr>
<tr><td>OS_GetLocalTime</td><td>0</td><td></td><td>0</td><td>Use error codes.</td></tr>
<tr><td>OS_SetLocalTime</td><td>0</td><td>0</td><td>0</td><td>Use error codes.</td></tr>
<tr><td>OS_TaskDelay</td><td>0</td><td></td><td>0</td><td>Use error codes.</td></tr>
<tr><td>OS_TimerCreate</td><td>0</td><td>0</td><td></td><td>Use error codes.</td></tr>
<tr><td>OS_unmount</td><td></td><td></td><td>0</td><td>Use error codes.</td></tr>
</table>

Table E.8: Summary of issues with magic numbers in OSAL

Function
OS_API_Init
OS_CountSemCreate
OS_initfs
OS_ModuleLoad
OS_mount
OS_MutSemCreate
OS_QueueCreate
OS_QueueGet
OS_SymbolLookup
OS_SymbolTableDump
OS_TaskCreate
OS_TaskDelete
OS_TaskRegister
OS_TaskSetPriority
OS_TimerAPIInit
OS_TimerCreate
OS_TranslatePath
OS_unmount

Table E.9: Summary of differences in output in OSAL