Software Fault Injection and Localization in Embedded Systems

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Ulrich Thomas Gabor

Dortmund

2021

Tag der mündlichen Prüfung:	25. Juni 2021
Dekan:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter:	Prof. Dr.-Ing. Olaf Spinczyk (Universität Osnabrück)
	Prof. Dr. Falk Howar (TU Dortmund)

ABSTRACT

Injection and localization of software faults have been extensively researched, but the results are not directly transferable to embedded systems. The domain-specific constraints applying to these systems, such as limited resources and the predominant C/C++ programming languages, require a specific set of injection and localization techniques. In this thesis, we have assessed existing approaches and have contributed a set of novel methods for software fault injection and localization in embedded systems.

We have developed a method based on AspectC++ for the injection of errors at interfaces and a method based on Clang for the accurate injection of software faults directly into source code. Both approaches work particularly well in the context of embedded systems, because they do not require runtime support and modify binaries only when necessary. Nevertheless, they are suitable to inject software faults and errors into the software of other domains.

These contributions required a thorough assessment of fault injection techniques and fault models presented in literature over the years, which raised multiple questions regarding their validity in the context of C/C++. We found that macros (particularly header files), compile-time language constructs, and the commonly used optimization levels introduce a non-negligible bias to experimental results achieved by injection methods operating on any other layer than the source code. Additionally, we found that the textual specification of fault models is prone to ambiguities and misunderstandings. We have conceived an automatic fault classifier to solve this problem in a field study.

Regarding software fault localization, we have combined existing methods making use of program spectra and assertions, and have contributed a new oracle type for autonomous localization of software faults in the field. Our evaluation shows that this approach works particularly well in the context of embedded systems because the generated information can be processed in real-time and, therefore, it can run in an unsupervised manner.

Concluding, we assessed a variety of injection and localization approaches in the context of embedded systems and contributed novel methods where applicable improving the current state-of-the-art. Our results also point out weaknesses regarding the general validity of the majority of previous injection experiments in C/C++.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[23] Ulrich Thomas Gabor, Simon Dierl, and Olaf Spinczyk. "Spectrum-Based Fault Localization in Deployed Embedded Systems with Driver Interaction Models." In: *Proceedings of the 38rd International Conference on Computer Safety, Reliability and Security* (*SAFECOMP '19*). Ed. by Alexander Romanovsky, Elena Troubitsyna, and Friedemann Bitsch. Turku, Finland: Springer International Publishing, 2019, pp. 97–112. ISBN: 978-3-030-26601-1. DOI: 10.1007/978-3-030-26601-1_7.

[24] Ulrich Thomas Gabor, Christoph-Cordt von Egidy, and Olaf Spinczyk. "Interface Injection with AspectC++ in Embedded Systems." In: *Proceedings of the 19th IEEE International Symposium on High Assurance Systems Engineering* (*HASE '19*). IEEE Press, Jan. 2019, pp. 131–138. DOI: 10.1109/HASE.2019.00028.

[25] Ulrich Thomas Gabor, Daniel Siegert, and Olaf Spinczyk. "High-Accuracy Software Fault Injection in Source Code with Clang." In: *Proceedings of the 24th IEEE Pacific Rim International Symposium on Dependable Computing* (*PRDC '19*). Pasadena, CA, USA: IEEE Press, Dec. 2019, pp. 75–84. DOI: 10.1109/PRDC47002.2019.00029.

[26] Ulrich Thomas Gabor, Daniel Siegert, and Olaf Spinczyk. "Software-Fault Injection in Source Code with Clang." In: *Proceedings of the 32th International Conference on Architecture of Computing Systems* (*ARCS '19*), *Workshop Proceedings*. 2019, pp. 1–6. ISBN: 978-3-8007-4957-7.

ACKNOWLEDGMENTS

Foremost, I thank the Technical University Dortmund for not only providing me a good education but also offering me multiple possibilities to gather the rare experience of giving lectures. In that context, I especially thank my supervisor Prof. Dr. Olaf Spinczyk. He has been a part of my way since the third bachelor semester and who has finally offered me the possibility to work in research and pursue this thesis. He not only enabled me to get a good insight into academic research but, at the same time, leaving enough room for independent evolvement.

Special thanks also to Prof. Dr. Falk Howar, Prof. Dr. Jens Teubner, and Prof. Dr. Peter Buchholz for agreeing to be part of the examination board, and Prof. Dr. Jian-Jia Chen for helpful mentoring comments along my way.

Additionally, I thank Dr. Horst Schirmeier for helping me with various questions and problems I encountered during my time. I also thank my colleague Hendrik Borghorst for having no clue regarding my problems but still trying to help me – and for becoming a very good friend. Many thanks also to Claudia Graute, who not only supported our whole chair by getting administrative tasks done but who took care of a lot of social events.

Furthermore, but not less important, I thank my parents, Anneliese and Wilfried Gabor, for supporting me not only during my thesis but throughout my whole life. They have always been encouraging regarding my choices and problems, and this helped a lot.

Additionally, there are a lot of friends and colleagues who helped me when times got more challenging, and I owe a lot to them (not necessarily in order of importance): Maurice, Glenn, Cordt, Heiko, Oliver, Thomas, Sebastian, Mirjam, Jonas, Jan Lukas, Alexey, Geoffrey, Matthias, Kuan-Hsun and Eva, Boguslaw, Markus, Georg, Alex, Daniel, Lea, Andreas, Michael, Benjamin, Torsten, Rolf, Mathias and Daniel.

Thank you all!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

AOP	aspect-oriented programming
API	application programming interface
AST	abstract syntax tree
DIM	driver interaction model
DSL	domain-specific language
FTAM	fault tolerance algorithm and mechanism
IR	LLVM intermediate representation
IoT	Internet of Things
MCSHS	method call sequence hit spectra
NLP	natural language processing
OCL	Object Constraint Language
ODC	orthogonal defect classification
OMG	Object Management Group
PDG	program dependence graph
POSIX	Portable Operating System Interface
ROM	read-only memory
RTTI	runtime type information
SBFL	spectrum-based fault localization
SBG	software behavior graph
SFI	software fault injection
SWIFI	software-implemented fault injection
UML	Unified Modeling Language
VCS	version control system

1 INTRODUCTION

As software increases in size and complexity, the number and complexity of software faults increase too – and often, the severity of caused failures rises as well. So far, developers and undergraduate courses usually concentrate on fault prevention and removal techniques at development time. Research, however, is also interested in so-called *residual* software faults [53], i. e., faults that elude the testing phase, and therefore most common practices against failures. The following two aspects are especially important:

1. to find and remove such faults, and
2. to assess the software to understand how it may fail in the field, to prevent failures in the field, and to compare the dependability amongst software.

This dissertation by publication deals with said field, with a particular focus on embedded systems, which come with additional limitations. Embedded systems are often smaller in terms of available resources and human-machine interfaces, i. e., they may comprise smaller processors and no display. Additionally, software for embedded systems is regularly custom-tailored to a specific need. This tailoring often involves the programming languages C/C++ and may be based on embedded operating systems, which can come without a runtime environment or are otherwise limited in functionality, for example, missing isolation between operating system and application software.

The thesis contributes solutions in two major fields:

1. the injection of software faults or fault effects called software fault injection (SFI) to better understand or improve a software system, and
2. the localization of already present faults in software, which eluded normal testing processes but may be found while observing the system in the field.

This chapter starts with the motivation for this research in Section 1.1, followed by the outline of this thesis in Section 1.2. It ends with a description of the contributions and the related publications to the current state-of-the-art in Section 1.3.

1.1 MOTIVATION

Much research effort has been spent in injecting and localizing faults in software for formerly so dominant systems like personal computers. However, rarely have methods been developed explicitly for embedded systems. With the Internet of Things (IoT), Industry 4.0 and more interconnected devices in general in an increasingly globalized world, the question is whether so far researched methods apply in the domain of embedded systems as well – or if entirely new and specific methods have to be developed for the more complex and ever more important embedded systems we see today.

Many software faults, even in this domain, can already be found at compile-time, e. g., Pathak et al. [57] developed an approach to find no-sleep energy bugs in Android, which are caused by pairs of on-off system calls where one

call is missing. Such faults, which are easily-reproducible, are often called Bohrbugs [30]. There have been efforts to combat such faults during development already. Modern compilers consist of a frontend to transform the programming language into an intermediate representation and a backend to generate an executable from this intermediate representation. LLVM is one of the well-known compilers of that type at the moment. Much effort has been invested into the LLVM compiler backend [43] to detect a variety of faults at compile-time with extended compiler checks, and researchers have developed additional static analyzers on top of the LLVM framework [62]. Clang, as a frontend for C/C++ in the LLVM project, provides a static analyzer, which has been regularly improved at Google's "Summer of Code" events[1] and extended by researchers [66]. There are developments outside of the LLVM project as well [8, 70]. However, developing and implementing an analyzer in a way that is easy to use and helpful turns out to be not an easy task [35]. From a developer's perspective, these analyzers may still lack a clear indication of the underlying fault (and how to fix it), or deliver an overwhelming amount of false positives.

Nevertheless, approaches at compile-time can often be applied to software for embedded systems as well, because they are run offline during development. On the other hand, many faults are hard to detect statically. Therefore, another field of research is dynamic checking, which executes (often modified) software and observes its behavior. One example is the well-known Valgrind [55] to find memory-related faults, but there are others as well [63, 68] – especially in the security domain [74], where overlooked faults often have severe effects, added canaries are supposed to protect against buffer-overflows.

In the context of embedded systems, additional peculiarities make it even harder to find bugs. The close relationship between hard- and software, the regular use of hardware registers or pointers, and the variation in the execution environments are just some examples. While often static analyzers can be applied to embedded software as well, the different execution environments bring challenges but also chances. This thesis will, therefore, concentrate on measures at runtime, i. e., fault injection and fault localization, which can be applied either during development or to a deployed embedded system in the field.

1.2 THESIS OUTLINE

This thesis by publication starts with a brief introduction of the basic concepts used throughout this work in Chapter 2. Chapter 3 presents our findings regarding SFI and is split into the well-known classes (i) injection of data errors, (ii) injection of interface errors, and (iii) injection of source code changes [54]. SFI into source code led to research regarding appropriate fault models, which is described in Chapter 4. The following Chapter 5 deals with the localization of software faults in deployed embedded systems. Chapter 6 contains the conclusion, potential future work, and a summary of all open research questions. Finally, reprints of the published articles this thesis builds on can be found after Chapter 6.

In the rest of the current chapter, the contributions provided by this dissertation are presented. This overview will give the reader a focused impression on the variety of contributions and novel ideas in contrast to the current state-of-the-art.

1 https://summerofcode.withgoogle.com/

1.3 MAIN CONTRIBUTIONS

The main contributions to the current state-of-the-art provided by this thesis have been presented in multiple publications. In this section, these publications will be presented arranged by the contributions to which they relate. All of the articles were presented at internationally recognized and peer-reviewed conferences and workshops on dependability and published in their proceedings. Additionally, this thesis contains some other ideas, which have not been published yet.

The developed methods were all implemented and are available as open-source tools in the interest of sustainability. Furthermore, these tools all come with tests, documentation, and were implemented according to current best-practices of software development. All these measures ensure that other researchers will be able to pick up and expand the work were needed.

1.3.1 *Fault Injection into Embedded Systems*

Each of the published articles concerning fault injection into embedded systems deals with one approach of the presented injection classes, i. e., injection at interface or source code level. The method for injections at interfaces makes use of AspectC++, whereas the method for injections into source code is based on Clang, as presented in the next sections. Additionally, this thesis discusses the idea of using aspect-oriented programming (AOP) and particularly AspectC++ for the injection of data errors and source code changes and why the idea has not been pursued further.

1.3.2 *Interface Error Injection*

Ulrich Thomas Gabor et al. "Interface Injection with AspectC++ in Embedded Systems." In: *Proceedings of the 19th IEEE International Symposium on High Assurance Systems Engineering (HASE '19)*. IEEE Press, Jan. 2019, pp. 131–138. DOI: 10.1109/HASE.2019.00028

The first publication deals with the injection of errors at interfaces, one of the injection classes we will discuss in this thesis. Previous methods to perform interface error injection were based on runtime modification by replacing dynamic libraries [7, 33]. Within the domain of embedded systems, such methods are often not applicable because there is often no runtime environment providing the function to load dynamic libraries.

Our approach utilizes AspectC++ to inject at compile-time without modification of the source code. This approach provides two advantages:

1. the developer does not have to modify the source code only to be able to perform injection campaigns, and
2. the approach works in any runtime environment.

The publication was a joint work with Christoph-Cordt von Egidy and Olaf Spinczyk. A prototype was developed as a Bachelor thesis by von Egidy. The author of this thesis conceived the underlying concept for the work, brought the software to a releasable state based on the developed prototype, and integrated the AspectC++ attribute functionality to further extend the application of the software to real software projects. The AspectC++ attribute functionality was developed as joint work with Olaf Spinczyk and Uriel Elias Wiebelitz.

1.3.3 *Injection of Source Code Changes*

Ulrich Thomas Gabor et al. "Software-Fault Injection in Source Code with Clang." In: *Proceedings of the 32th International Conference on Architecture of Computing Systems (ARCS '19), Workshop Proceedings*. 2019, pp. 1–6. ISBN: 978-3-8007-4957-7

Ulrich Thomas Gabor et al. "High-Accuracy Software Fault Injection in Source Code with Clang." In: *Proceedings of the 24th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '19)*. Pasadena, CA, USA: IEEE Press, Dec. 2019, pp. 75–84. DOI: 10.1109/PRDC47002.2019.00029

These publications are about fault injection of code changes into source code, another class of approaches for SFI. Such injections can be performed at different abstraction layers: into assembly [13], into LLVM intermediate representation (IR) [28, 41], and directly into source code [53]. To be able to perform fault injections, it is necessary to have a model of faults. In the past, multiple publications gathered information to define fault models, which were supposed to represent real-world faults [4, 15, 17, 27, 29, 39, 40, 53, 61]. With injections into source code, it should be easiest to simulate a programmer's fault accurately compared to the chosen fault model.

Our work shows that, despite these past efforts, it is at least questionable if injections were really performed according to the fault model. Most developers use the optimization level `-O2`, which is the recommended optimization level (e. g., by Debian[2]), and which can result in quite different binaries from the original. Injections performed on the binary level are, therefore, subject to biases. Furthermore, macros and C/C++ compile-time language constructs can have an enormous effect because of their (nearly) Turing-completeness [69]. In the publications related to this class of SFI, we presented a new Clang-based injection approach, which performs injections closer to the selected fault model.

Furthermore, the methods presented in literature often come with additional support code to perform the injections [28, 41]. Additional code might come with additional dependencies or compiler requirements, which can pose a problem in the context of compilers for embedded systems. Also, additional code increases the size of the image, which can be a problem for devices with only limited available memory. Our approach takes these constraints into account and is therefore especially useful in the context of embedded systems.

This was a joint work with Daniel Siegert and Olaf Spinczyk. An already existing prototype by the author of this thesis was finished as a Bachelor thesis by Daniel Siegert. The author of this thesis brought the software to a releasable state and contributed most of the evaluation, assessing the accuracy of the injection tool regarding the fault model.

2 https://www.debian.org/doc/debian-policy/ch-files.html

1.3.4 *Program Spectrum-based Fault Localization*

Ulrich Thomas Gabor et al. "Spectrum-Based Fault Localization in Deployed Embedded Systems with Driver Interaction Models." In: *Proceedings of the 38rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '19)*. Ed. by Alexander Romanovsky et al. Turku, Finland: Springer International Publishing, 2019, pp. 97–112. ISBN: 978-3-030-26601-1. DOI: 10.1007/978-3-030-26601-1_7

While the previous contributions deal with the introduction of faults into software, this one regards the localization of bugs already present in a software. One approach to locate faults is spectrum-based fault localization (SBFL), where the so-called spectrum is used to pinpoint a fault. A spectrum encompasses execution information, e. g., the executed functions.

We have brought SBFL to embedded systems with the idea of using an isolated additional embedded system to monitor the observed system infield. Furthermore, we evaluated multiple spectrum-types and suspiciousness metrics in the real-world scenario of a simulated combustion engine. For the autonomous assessment of runs as failing or succeeding, we also developed a new type of oracle called driver interaction model (DIM).

The resulting publication was a joint work with Simon Dierl and Olaf Spinczyk. Simon Dierl developed the software in the context of his Master thesis. The author of this thesis conceived the concept, especially regarding the novelty to execute this approach on a second embedded system, and the underlying idea to use the interaction between soft- and hardware as an oracle.

1.3.5 *Transformation of Model Constraints to Assertions*

Additionally to the main contribution of spectrum-based fault localization, we discuss the implementation of a fault localization approach based on Object Constraint Language (OCL) in C++. With OCL annotated Unified Modeling Language (UML) diagrams should be automatically transformed to AOP code, which then can check the validity of the OCL rules at runtime and report deviations. While the idea to transform OCL to AOP is not new [18], we additionally explain why an implementation in AspectC++ should be possible, highlight potential pitfalls and assess existing solutions regarding their applicability in the context of embedded systems.

2 BACKGROUND

The current chapter gives the reader an overview of the used terminology and provides some background regarding the topics of this thesis.

The first Section 2.1 introduces concepts of dependability and basic terminology. Section 2.2 then presents perspectives of faults and possibilities to model them, primarily for the use of software fault injection. In Section 2.3, basic concepts of software fault injection (SFI) and especially its applications are described. Since some concepts of this thesis are based on AspectC++, Section 2.5 gives an introduction to aspect-oriented programming (AOP) and AspectC++. The chapter closes with an overview of the specialties of embedded systems in comparison to desktop computers or servers in Section 2.6

2.1 BASICS OF DEPENDABILITY

A standard taxonomy regarding dependability has been presented by Avižienis et al. [2], and this thesis makes use of the terms as presented in that article. In this context, the three terms fault, error, and failure are of topmost importance. Their relationship in the form of a chain is shown in Figure 2.1. A failure is defined as a deviation of a service from its expected behavior, which can manifest, for example, in unavailability or wrong responses. In the context of this thesis, this can be, for example, a crash of a software system or wrong messages being sent to other components. While a failure is observable, it is caused by a deviated state of the system which we call error and which is not observable from outside the system. For software systems, one example of an error is unexpected modification of memory contents. The underlying cause of an error is called fault. In software systems, this is what is colloquially known as a bug and can, for example, be wrong handling of input or a deviation between implementation and intended function of an algorithm. A fault can be triggered by a failure of another component or system; therefore, multiple chains can be connected.

Faults are present in nearly all software systems but manifest differently depending on the system and their location. Most readers will be familiar with faults that they introduced themselves as a deviation between what should be implemented and what was implemented. Even in software that was proven to be implemented correctly, there is potential for faults, for example, a deviation between the formalized functionality and what should have been formalized.

It is evidently necessary to deal with the presence of faults, which can be done at different stages of the software development process. A short overview of the related terminology is shown in Figure 2.2. Fault prevention means all efforts to avoid the introduction of faults in the first place, for example, by training developers. Fault removal is aimed at the removal of faults, for exam-

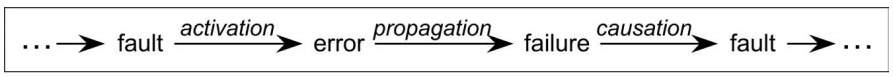

Figure 2.1: The dependability chain showing the interaction between faults, errors and failures [2].

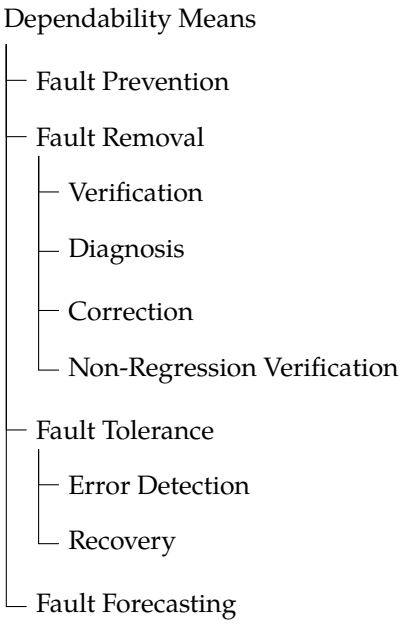

Figure 2.2: Overview of dependability means based on the taxonomy of Avižienis et al. [2].

ple, by utilizing compiler checks or test processes. Since it is often impossible to get rid of all faults, fault tolerance aims at avoiding failures, for example, by introducing redundancy and voting mechanisms [42]. Lastly, it may be of interest to anticipate how many failures and what deviating behavior are to be expected, which is known as fault forecasting.

2.2 FAULT MODELS FOR SOFTWARE FAULT INJECTION

For multiple of the dependability means mentioned earlier, it is helpful to simulate the presence of faults, which is the goal of SFI. For that, it is necessary to have an adequate model of software faults. This fault model should be as close to real software faults as possible, so that evaluation results are trustworthy.

One possible distinction of different types of faults is shown in Figure 2.3 [30, 53]. In general, the fault space can be divided into residual and non-residual faults. On the one hand, residual faults regularly occur in the field and are hard to find during development. On the other hand, non-residual faults can be easily found while testing. Whereas residual faults are often Mandelbugs, i. e., bugs which are hard to reproduce, for example, because they require complex environment conditions to be activated, non-residual faults are often easy to reproduce and are also called Bohrbugs [31]. The former are hard to detect at development time, whereas the latter can often be found through testing.

For both types of faults, research has parted ways regarding their methods and goals. On the one hand, research related to non-residual faults, which can be easily found with testing, concentrates on mutation testing. The latter is an approach to improve the test coverage of software by introducing changes to it and check if any test notices this modification. If this is not the case, this is taken as a sign that the tests should be improved to spot such a change. On the other hand, research regarding residual faults concentrates on SFI, introducing faults which are hard to detect at development time.

Based on this distinction, the methods dealing with non-residual faults can rather be classified as fault avoidance mechanisms, whereas the methods to deal with residual faults can rather be classified as fault tolerance mechanisms.

It must be said though that such classifications are never perfect. In practice, it can happen that a simple fault, which could have been found by tests, was not found and only showed itself in the field after deployment. Just the fact that one cannot know in advance which faults are present in the software and therefore are to be expected is reason enough to question any classification which promises perfection.

In the domain of embedded systems, it is possible to explain potential faults more concretely. Possible faults are various, though, because the range of systems that are classified as "embedded" is wide, for example, interconnected microcontrollers in cars, systems controlling the production in factories (Industry 4.0), and devices in the Internet of Things (IoT) domain are all called embedded systems.

First, often there is some form of basic software in an embedded system, i. e., an operating system or surrounding library code, which can be shared across different systems. Such basic software is susceptible to all known faults in such software systems, for example:

- accessing wrong memory locations,
- insufficient isolation between applications and basic software, and
- unfair assignment of CPU (or other resources) to applications, maybe leading to starvation.

Second, there are regularly applications, which provide the desired functionality of the embedded system. These applications can suffer from a variety of potential faults known from regular software development, for example:

- buffer overflows,
- unexpected input values, and
- wrongly implemented algorithms.

However, the variety is regularly more restricted compared to other software systems. Embedded systems must often fulfill critical tasks and are, therefore, often subject to firm programming practices. For example, MISRA-C is a standard coming from the automotive domain, restricting the usable language features of the programming language C [50]. Strict application of these rules will likely erase some classes of faults. Which faults are erased depends on the concrete programming practices applied.

Third, embedded systems often interact with the surrounding environment, either by reading sensor data, by communicating with other (embedded) systems, or by controlling physical mechanisms. This peculiarity enables an additional range of potential software faults, which may be unknown to software developers not familiar with this domain, for example:

- wrong handling of unreliable communication channels,
- missing mechanisms to cope with broken attached hardware, and
- incorrect interaction with periphery.

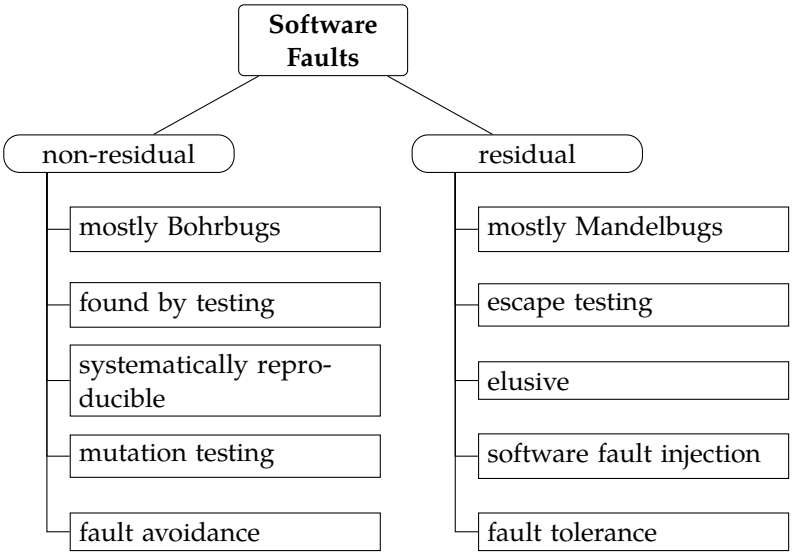

Figure 2.3: Two types of software faults and their characteristics.

Even though it is now clear what faults a fault model should represent, it is still unclear what a fault model must contain, so that it can be used for the injection of faults. For SFI, it is essential to know what modifications to perform and what frequency to use. For example, it may often be the case that an IF statement plus some source code lines are missing. A tool can quickly generate all possible modifications of that type for a piece of software. If one wants to sample from the available fault space due to time constraints on the runtime of experiments, which is often the most problematic part of such an assessment, it may be desirable that the sample has the same distribution as real software faults.

Both parts of a fault model – types and distribution of faults – can only be approximated by assessing what is known about faults. Some researchers have examined bug reports of large software projects, classified the required changes by hand, and then created models that are usable by others [4, 17]. Despite the efforts, it can often be shown that such models are not as representative as one might desire [39]. It is not proven but likely that the different types of software, software development processes, involved human developers, and even the people involved in modeling a fault model have a significant impact on the representativeness of the fault model.

2.3 BASICS OF SOFTWARE FAULT INJECTION

SFI has been introduced as a method for multiple use cases. The following three are currently known and relevant:

FAULT REMOVAL FROM FTAMS: This application aims at improving a fault tolerance algorithm and mechanism (FTAM) mostly during development by qualitatively using the results of an injection campaign to improve its implementation.

DEPENDABILITY BENCHMARKING: Dependability benchmarks help to compare the dependability properties of multiple components or systems. A benchmark should fulfill multiple prerequisites to be helpful, for ex-

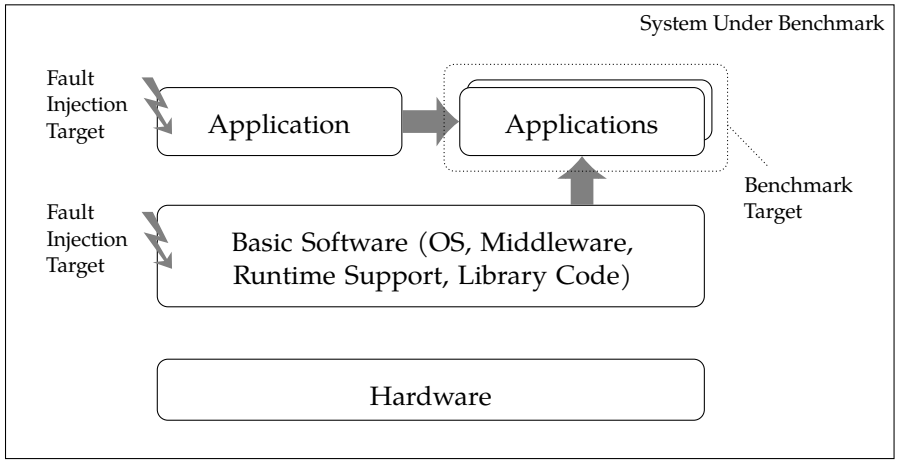

Figure 2.4: Benchmarking of Application-Level Software, adopted from Natella et al. [54].

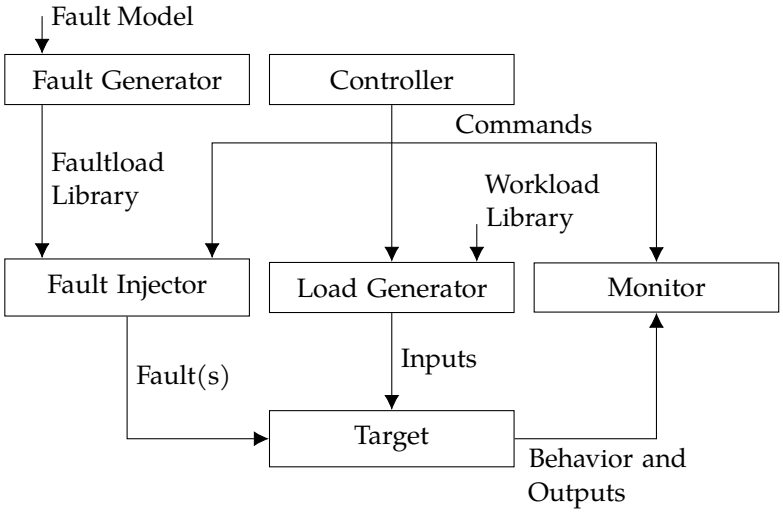

Figure 2.5: Conceptual components of fault injection, adopted from Hsueh et al. [34] and Natella et al. [54].

ample, precise experimental steps that support the reproducibility of results.

FAULT FORECASTING: This use case encompasses quantitative evaluations of fault-tolerance properties of systems, often based on probability theory, e. g., Markov models. For example, it might be of interest how much delay is caused by an FTAM and how often this is to be expected.

All three use cases have in common that the underlying idea is to inject faults into those components which are currently not under review. Figure 2.4 shows an example where faults are injected either into applications running on the system or the underlying system software, causing an effect on the benchmark target. In case of fault removal from an FTAM, faults are injected into the software while the FTAM is under review.

Furthermore, all SFI experimental setups follow a similar pattern, which is shown in Figure 2.5. First, a faultload library must be created. In the context of fault injection into source code, this could be a list of patch files, each emulating

a faulty situation. During an experiment, a controller controls an injector component, a load generator, and a monitor. The fault injector component introduces the faulty situation into the target-under-test. The load generator generates the inputs to run the software, for example, some binary input. All information regarding the experiment is gathered by the monitor, e. g., the introduced fault, the workload, and the behavior of the software. It can then generate a report for further evaluation of the injection campaign.

Approaches regarding SFI can be split into three classes:

INJECTION OF DATA ERRORS Corrupt memory or registers similar to hardware fault injection.

INJECTION OF INTERFACE ERRORS Corrupt input and output values of component interfaces, be it inside a piece of software or at interfaces to other systems, i. e., other hard- or software.

INJECTION OF CODE CHANGES Corrupt code to closely emulate programming bugs.

There exist various methods to perform injection in each of the classes. For example, injection of code changes can happen on multiple abstraction layers:

- into source code,
- into intermediate code of a compiler, or
- into binary code.

The approaches have different properties. For example, on the binary level, language constructs of the programming language must first be found, which can be a difficult task, but modifications on this level are fast because they do not require recompilations for each experiment. On the other hand, injections into source code are more straightforward, but for each new experiment, the software must be recompiled, which can take much time.

Considering the domain of embedded systems, all three use cases for SFI are relevant. An FTAM should regularly be in place to allow the software to cope with degrading periphery or software faults. The removal of faults from such mechanisms is important to improve the robustness of the system. Benchmarking the dependability of such systems is necessary, for example, when multiple software systems are available to fulfill a task. Fault forecasting allows to assess the potential damage a malfunctioning system can cause, especially when it is interconnected with other devices or physical mechanisms.

The available classes of SFI can be used to emulate various faults in an embedded system, see Section 2.2 for details regarding faults in embedded systems. Injection of data errors can be used to simulate broken hardware, for example, when the periphery is mapped into the memory, and therefore be used to remove faults from an FTAM. Injection of interface errors can be used to simulate faults crossing components of the systems, for example, from the application to the system software and vice versa or passing of messages to interconnected devices or periphery. This may be especially helpful when comparing the dependability of software components. Injection of code changes can be used to emulate a programmer's faults.

However, we will later see that most approaches of SFI have been used to inject multiple types of faults. There is no clear guideline stating which class of SFI to use when emulating a specific fault type.

2.4 SOFTWARE FAULT LOCALIZATION

Whereas the previous sections dealt with the introduction of faults, it is also of interest how to localize faults that are already present in software. There are numerous localization techniques [73]:

PROGRAM LOGGING: Developers add instructions to output logging information. These instructions are often removed after a bug has been localized.

ASSERTIONS: Assertions are statements that check specific conditions at runtime and stop execution otherwise. They can be used, for example, to check invariants.

BREAKPOINTS: Breakpoints are statements that stop the execution at a specific code position to allow a developer to examine the current state.

PROFILING: Profiling uses runtime metrics, such as memory usage, to detect unexpected behavior.

SLICE-BASED TECHNIQUES: These techniques try to remove parts of software, so that it still behaves like the original software in some aspects but is easier to understand.

SPECTRUM-BASED TECHNIQUES: A spectrum comprises specific runtime information, for example, the list of executed functions. Together with information on whether the corresponding execution succeeded or failed, the approach tries to identify suspicious parts of the software, which may be responsible for failing executions.

STATISTICS-BASED TECHNIQUES: Predicates are added to the software, and with statistical mechanisms, a score between failures and predicates is computed. These scores should help a developer pinpoint the responsible fault.

STATE-BASED TECHNIQUES: The idea here is to use the state of a program, i. e., values of variables, to debug programs. One possibility is, for example, to compare the state with a reference state to recognize deviations.

MACHINE LEARNING-BASED TECHNIQUES: Machine learning can also be used to localize bugs. One possible technique involves the training of a back-propagation network with coverage data of test cases and their execution result. Later, virtual test cases, each only covering a single statement, can be used so that the output is the probability that the corresponding statement contains a bug.

DATA MINING-BASED TECHNIQUES: Due to the sheer amount of data that can be gathered when tracing bugs, e. g., multiple complete execution traces, data mining techniques may be helpful to detect patterns, e. g., executed statements, which lead to a failure.

MODEL-BASED TECHNIQUES: Models can be of multiple uses to localize bugs. For example, if a model of the correct behavior of software is available, it can be used to spot differences when executing a faulty implementation. On the other hand, it is also possible to create models of a faulty software to then identify model components, which may be responsible for an observed failure.

This thesis makes use of techniques coming from the areas of assertions, spectrum-based, and model-based techniques.

Listing 2.1: Example advice tracing the signature of each invoked function.

```
aspect Tracing {
    pointcut match() = "bool %::%(...)" || derived("MyClass");
    advice execution(match()) : before() {
        cout << JoinPoint::signature() << endl;
    }
};
```

2.5 BASICS ON ASPECT-ORIENTED PROGRAMMING (AOP)

Part of this thesis is the assessment of how useful AOP language features are for fault injection processes. The current section will give an overview of AOP and its available implementations.

The goal of AOP is to provide language features, which allow a structured implementation of so-called crosscutting concerns [21]. Crosscutting concerns are those which affect multiple functions of a software similarly, for example, error detection and correction mechanisms. While previously, it was necessary to respect such details at all necessary source locations, with AOP, it is possible to gather this functionality in a single implementation artifact and specify where it should apply. This is not restricted to the execution of code, but can also affect data structures; for example, it is possible to add member variables to classes or execute code when accessing variables.

Such a language extension can help solve various problems. For C++, there exists the AspectC++[1] [64] language extension, which was used in many articles [5, 6, 22, 45–47], especially regarding embedded systems and software product lines, i. e., software that can be tailored at compile-time to multiple use cases. For Java, there exists an implementation called AspectJ[2] [38], which formed the basis for many research articles regarding AOP features and software architectural difficulties [9, 16, 32, 38, 67].

Besides endianness and software product lines, there are also various other application scenarios for AOP. Examples include locking, synchronization, tracing, character set settings, and caching, only to name a few.

Listing 2.1 shows an example tracing aspect written in AspectC++ that adds tracing output to C++ functions that return a boolean value or methods of a class that is derived from a class with the name `MyClass`. The general structure is similar to how a class is implemented in C++, but for aspects, the keyword `aspect` is used to introduce a new crosscutting concern and, instead of functions, code to be executed is gathered in advices introduced with the keyword `advice`. The part after the `advice` keyword describes where and when the functionality should be applied to and can also be outsourced for reusability in other advices with the keyword `pointcut`. The code inside of advices can make use of the built-in object `tjp` and the built-in class `JoinPoint` to get information regarding the current location which is augmented, i. e., the source code line number or the return type of the augmented function. AspectC++ therefore provides introspection extensions to C++ additionally to, for example, `type_traits` and runtime type information (RTTI).

AOP and AspectC++ provide even more functionality than we use throughout this thesis. While in this thesis the attribute functionality of AspectC++ was only used to specify pointcuts, Friesel et al. have shown that AspectC++ can be

1 https://aspectc.org
2 https://www.eclipse.org/aspectj/

used to replace non-portable C++ attributes and how portable attributes can be extended with domain-specific knowledge, and that AspectC++ can support co-development of source code and corresponding models [22]. AspectC++ can also affect C++ templates [45] and can be used for whole-program analysis [6]. The order in which aspects are applied to a pointcut can be specified in AspectC++ and AspectJ. Both language extensions also provide pointcut functions to select pointcuts based on the current control flow at runtime.

2.6 SPECIALTIES OF EMBEDDED SYSTEMS

Injecting and localizing faults or errors in software for embedded systems pose some difficulties not relevant for injections and localization in other types of software. The current section will give an overview of potential differences, although it must be noted that the field of embedded systems is vast and different embedded systems have different specialties.

In general, development for embedded systems requires the use of embedded operating systems. These often provide little to no runtime environment, while, for example, most of the proposed methods for interface injection require dynamic libraries and operating system functions to replace them. Additionally, often not only the software environment poses limitations but the hardware as well. Hardware limitations may include smaller processors, less memory, missing displays, and other human-machine interfaces.

While some of the specialties, particularly regarding access to the system, can be circumvented when using a simulator, the current trend seems to move development for embedded systems away from simulators and onto so-called development boards. Texas Instruments has declared that it would move away from simulators to low-cost development boards, which affects, for example, the well-known MSP430 family of microcontrollers. We were not able to find information regarding a simulator for the other microcontroller used during working on this thesis, the Wandboard Quad, which hosts an i.MX6 CPU based on the ARM Cortex-A9 instruction set. Apple provides a simulator for its operating system iOS, which is used on the iPhone and iPad. However, although the ecosystem is already quite large in comparison to systems considered more embedded, even this simulator can only be used for rudimentary development. It does not simulate all components real hardware provides – e. g., no camera support, no Bluetooth, no sensors – nor all operating system functionalities – e. g., no push notifications, no framework support. Google provides an emulator for Android, but it also has some limitations – e. g., no Bluetooth, no NFC, no SD-card support, no USB.

For the simulation of faults or errors on real hardware, injections must be performed into the executable or via hardware mechanisms during runtime, e. g., breakpoints. Localization of faults in software deployed on real hardware requires that either information must be processed on the embedded system itself, which may pose a problem given only limited computational resources, or must be transmitted to another system with additional effort.

In general, investigating an embedded system can be complicated because it may not be possible or desirable to run additional software to inspect the current status, including an operating system. One alternative is to examine its interaction with its surroundings, e. g., peripheral devices. Another possibility can be to use development mechanisms of the development board, which, for example, may allow accessing the current contents of the memory.

Additionally, it may not only be a problem to investigate an embedded system, but often the embedded system as a whole, including its operating system

and applications, is tailored to its particular needs. This tailoring may result in no isolation between the operating system and applications. A malfunctioning application may then draw the whole system unusable and make it hard to investigate the current status of the system after injecting an error.

Concluding, there are multiple potential pitfalls when applying known methods to embedded systems, which we will discuss in this thesis. Furthermore, we develop new approaches when necessary, such that fault injection and localization can happen in embedded systems with their given specialties.

In summary, this chapter has given an overview of basic terminology regarding dependability, fault models, fault injection, and localization. Additionally, AOP has been introduced briefly, as it is used throughout this thesis, and the specialties of embedded systems have been discussed, which render the applicability of known approaches complicated.

3 SFI FOR EMBEDDED SYSTEMS

The first main contribution of this thesis deals with software fault injection (SFI), especially in the context of embedded systems, which requires special care because of additional limitations regarding runtime environment, programming languages in use, and available computational power and memory, as laid out in Section 2.6.

In such a restricted environment centered mainly around the programming languages C/C++, the language extension AspectC++ is useful for solving various problems, as presented in Section 2.5. It is immediately apparent that aspect-oriented programming (AOP) can be used for the injection of errors at software interfaces. In contrast, it is not apparent if AOP can be used for injections according to the other classes as well and how beneficial the use of AOP is.

The first part of this chapter laid out in Section 3.1 will, therefore, assess AOP across the multiple classes of SFI approaches to answer how useful AOP and particularly AspectC++ are for fault injection methods. The injection of errors at interfaces with AspectC++ constitutes one of the major contributions of this thesis. The second part of this chapter in Section 3.2 then focuses on the injection of code changes based on mechanisms Clang provides and is another major contribution of this thesis.

Two of the topics are covered in the following articles. We expect the reader to know its contents for the rest of this chapter, but we will give summaries at appropriate points during this chapter. Additionally, we will evaluate further ideas in this chapter, which did not fit into either of the articles.

Ulrich Thomas Gabor et al. "Interface Injection with AspectC++ in Embedded Systems." In: *Proceedings of the 19th IEEE International Symposium on High Assurance Systems Engineering* (*HASE '19*). IEEE Press, Jan. 2019, pp. 131–138. DOI: 10.1109/HASE.2019.00028

Ulrich Thomas Gabor et al. "Software-Fault Injection in Source Code with Clang." In: *Proceedings of the 32th International Conference on Architecture of Computing Systems* (*ARCS '19*), *Workshop Proceedings*. 2019, pp. 1–6. ISBN: 978-3-8007-4957-7

Ulrich Thomas Gabor et al. "High-Accuracy Software Fault Injection in Source Code with Clang." In: *Proceedings of the 24th IEEE Pacific Rim International Symposium on Dependable Computing* (*PRDC '19*). Pasadena, CA, USA: IEEE Press, Dec. 2019, pp. 75–84. DOI: 10.1109/PRDC47002.2019.00029

3.1 PRACTICALITY OF AOP FOR SFI

This section discusses the practicality of AOP for SFI. An introduction to AOP was already given in Section 2.5. Since AOP allows separation of concerns and augmentation of runtime execution, it can be used to modify the behavior of software without the need to modify the source code. Additionally, because aspects are woven in by the compiler, it requires no runtime support and is an excellent candidate to work in the restricted environment of embedded systems.

The rest of this section is parted into three subsections, each dealing with one of the classes of SFI and the possible benefits of using AOP to inject corresponding faults or errors.

3.1.1 *AOP for Data Error Injection*

Data error injection is the first class of SFI, and is the closest to hardware-fault injection. Actually, first approaches originate in software-implemented fault injection (SWIFI) for hardware faults. The goal of data error injection is to emulate the effects of faults by modifying the content of the memory before or during the execution of a program.

C++ is a language that grants much freedom to access the memory. Therefore, assessing AspectC++ for the injection of data errors makes sense. While it is possible with C++ and, therefore, also with AspectC++ to modify its own code during execution of a program, this approach comes with multiple problems:

1. In the context of embedded systems, the software is typically stored in read-only memory (ROM) and is therefore not modifiable at runtime. Nevertheless, it is possible to use `get()`/`set()` advice of AspectC++ to redirect data access and modify it at runtime, but this incurs non-negligible overhead and may therefore be not desirable.
2. As we have shown [25], it is hard to emulate software faults on that level of abstraction correctly. For example, it is hard to find patterns matching a specific structure of the source code.
3. Modifying memory directly requires an understanding of the encoding of data types to bytes and the architecture, which often requires cumbersome architecture-specific coding for multiple architectures.

Other features provided by AOP do not improve this situation, as they are directed at higher levels of abstraction, i. e., the separation of concerns of implementation artifacts. Therefore, and since other options already exist [3, 37, 54], we did not pursue this path.

3.1.2 *AOP for Interface Error Injection*

Injecting interface errors into software is used to simulate errors in inputs and outputs of components. In the context of software, these can be parameters or return values of functions, methods, or interactions with hardware. It is regularly used in robustness testing, which tests to which degree software operates correctly in the presence of invalid inputs or faulty environments.

The injection of interface errors is possible with AOP as it provides access to parameters and return values of functions and methods. A simple example of a possible injection implemented in AspectC++ is shown in Listing 3.1, which is a concept of the approach we have contributed with this thesis and the corresponding article [24]. The advice code redirects each call to the function `myfunction`, which returns a value of type `int` and takes arbitrary arguments, so that the return value and parameter are processed by a provided function `inject`. The `inject` function checks for an activation trigger and, if an error should be injected, it modifies or replaces the return value or parameter.

Our contribution, which is available as an open-source implementation[1], provides further functions around this underlying mechanism. Foremost,

1 `https://ulrichgabor.de/r/aofit`

Listing 3.1: Basic example of injecting errors in return value and parameter.

```
advice call("int myfunction(...)") && result(resultvar)
     : after(int * const resultvar) {
    inject(tjp, resultvar);
}
advice call("void myfunction(int)") && args(a)
     : before(int * const a) {
    inject(tjp, a);
}
```

we solved multiple issues regarding the transformation of given interface specifications in JSON syntax into automatically generated aspects. Automatic transformation is not only convenient but allows for the easy application of injection campaigns at interfaces conforming to the Portable Operating System Interface (POSIX) standard without knowledge of AOP.

Regarding the domain of embedded systems, we also depicted and evaluated multiple methods to perform injection experiments, which is a non-trivial task given that software must often be run on isolated development hardware. The most straightforward variant is to flash a patched image to the hardware for each new experiment. Other variants to reconfigure an image for the next experiment include GDB – the GNU debugger – or arbitrary protocols which are connected via serial port, ethernet, or JTAG, which is an industry-standard for debugging embedded systems. These approaches improve the time required for an injection campaign because flashing can be a time-consuming process. Details can be found in our publication [24].

We compared our approach to state-of-the-art approaches, which use dynamic library interception to inject errors at such interfaces, for example, `LD_PRELOAD` [7, 33]. While this mechanism is comfortable, it can only inject errors at interfaces to dynamic libraries. Since embedded systems often make no use of dynamic libraries, this mechanism cannot be applied to them. We counted interfaces of some standard software and found that regularly the number of interfaces of dynamic libraries is only around 5 % to 10 % in comparison to the total number of interfaces. Therefore, up to 90 % of interfaces are left, which cannot be injected with approaches based on dynamic libraries.

Not all interfaces may signal errors. We, therefore, explicitly considered those interfaces again that return a pointer or integer and can, in theory, return an error. We found that for a binary of size 20,732 kB, there are 97 interfaces returning a pointer and 34 interfaces returning an integer value. Instrumenting all of them results in a binary of size 22,628 kB and, therefore, a growth of 9 % in size, which seems appropriate.

Concluding, we have contributed a new method to inject errors at arbitrary interfaces. We have taken special care for embedded systems and have depicted multiple variants to perform injection campaigns efficiently and to generate campaigns automatically based on a POSIX specification.

3.1.3 *AOP for Injection of Code Changes*

The previous section has shown that AOP and particularly AspectC++ is an excellent addition to the C++ language to perform interface error injection, but the current section will show that AOP and AspectC++ cannot be used

to inject code changes in its current form – and why this also might not be desirable.

Code changes and programmer faults are syntactical differences in comparison to correct code. Often, it is hard or even impossible to specify what semantical change follows from a syntactical change; see Rice's theorem [60]. Therefore, staying on the syntactical level to emulate software faults is the preferred way.

AOP languages, in comparison, do not work on the level of syntactical structures but a semantical level. They allow for augmentation of code by specifying pointcuts and code that should be executed at the pointcuts or modify data structures. Such pointcuts are specified in a syntax-agnostic language; for example, they allow to augment function calls filtered by the current control flow. Besides filtering, there is no way to tell how a function was called, e. g., normally, via a pointer or reflection mechanism, and it is not of relevance for the semantic level AOP languages operate on.

Despite this fundamental discrepancy between the level AOP operates on, and the level injections are performed, we assessed whether it is possible to inject code changes with AspectC++, as one of the prominent AOP language extensions used in embedded systems. For the selection of potential source locations to inject faults at, AspectC++ provides pointcut functions to match accesses to global variables with `get()`/`set()` and uses of built-in operators, e. g., +, with `builtin()`.

At first glance, it is, therefore, possible to inject different behavior in multiple positions in the code, at least for some fault types. However, a closer look reveals that the pointcut functions for variables currently work only with global variables and members of classes. Local variables cannot be augmented, which is essential for modifying basic local algorithms, e. g., a sorting algorithm. This limitation is, to our best knowledge, of technical nature, so it could be solved by changing the AspectC++ parser. However, the primary purpose of AOP is to implement crosscutting concerns, and these are often not affected by local variables. Therefore it is unclear if there are enough good reasons to implement pointcut functions for local variables.

Another problem is that context is required to perform injections. Consider a matched built-in operator `&&` which may be subject for an injection of the type "missing AND condition in branch condition", see Table 4.2 on page 25 for an early overview of potential fault types. Only with context, i. e., the types of the parent nodes, it is possible to determine if the built-in operator is used in a branch condition or if it is part of an assignment. The joinpoint API of AspectC++ provides only basic information regarding the structure of the code. Among the ones seeming most helpful at first glance are `JPID`, `JPTYPE`, `*signature()`, `*filename()` and `line()`.

However, all this information still does not provide enough context to perform accurate injections. Consider, for example, a minified C++ file, which often boils down to only one line of source code, then `*filename()` and `line()` are not helpful anymore. The signature of a method provides no context for built-in operators at all. The only remaining possibility is `JPID`, an id unique to each joinpoint. One solution to acquire the required context based on this feature is a technique used for more complex reflection tasks in AspectC++. The idea is to integrate the information AspectC++ gathered during a first parsing pass into the aspects and then compile the final executable. This technique could be used to obtain necessary information for each `JPID`, e. g., the types of the parent nodes of the currently matched joinpoint, which could then be integrated into the final executable. Unfortunately, this idea entails the problem of

identifying the type and context for each token of the source code. Since this is not possible with AspectC++, otherwise we could use this mechanism directly, one would have to utilize another tool to get this information, i. e., some kind of parser. Now the question is, why implement a parser to obtain this context information, and then integrate this information into aspects, when the new software could just inject the fault itself? Considering that the motivation to use AspectC++ for fault injection is not to need to use specialized software, this idea seems flawed.

Considering all these problems and the difference between syntactical fault injection and the semantical level AOP operates on, we instead concentrated on developing a method for the injection of code changes, which is not based on AOP and AspectC++.

3.2 INJECTION OF CODE CHANGES WITH CLANG

While injection of code changes with AspectC++ is not easily done, as shown in the previous section, the modern compiler frontend Clang for C/C++ languages, based on the LLVM backend [43], provides features that are useful for the implementation of an injection method.

Foremost, it provides a domain-specific language (DSL) to specify abstract syntax tree (AST) Matchers[2], which allows coarse-grained matching of AST nodes. Together with callback functions that have fine-grained access to the AST nodes, it is possible to match exactly those syntactical structures that need to be found for the used fault model. Furthermore, it provides rewriting mechanisms of source code, which allow for the injection of faults according to the fault model.

We used these techniques to implement clang-sfi[3], a software-fault injection method based on Clang [25, 26]. Our publications show that the use of Clang and the described features already led to a significant improvement regarding the accuracy of the tool. In this context, accuracy means the proximity between generated faults to inject and the faults which should have been injected according to the underlying fault model. We further improved the accuracy by allowing to inject faults into macros, which is normally not possible with the Clang rewriter functionality, and by respecting headers and C++ project structures.

A qualitative examination of multiple examples shows that neglecting these C++ peculiarities can lead to a drastic rift between the desired injections and the actual injections, which inevitably must have an impact on the observed failures for affected projects. However, such a quantitative evaluation is still future work, see Section 6.1.

If the impact is as drastic as envisioned, it might render many results of other articles questionable. For example, comparisons between the injections inside components and at interfaces of components [52] are based on injections on the binary level, which we have shown to be not accurate in many current development scenarios. Actually, all results which are based on injections on lower levels than source code and which are generalized to arbitrary software compiled in modern development environments should be viewed with caution. Additionally, injections into the source code of projects making heavy use of headers or macros are likely biased.

The latter is especially severe in the context of embedded systems, where programming often makes heavy use of macros and transformations of macros

2 https://clang.llvm.org/docs/LibASTMatchersReference.html
3 https://ulrichgabor.de/r/clang-sfi

to C++ source code regularly require features of new C++ standards, which might not be implemented by compilers for embedded systems.

In this chapter, we have discussed three classes of fault injection into embedded systems: data error injection, interface error injection, and injection of code changes. Notably, we have contributed insights on how useful AOP is for each of these classes and concluded that it is most helpful for the injection of interface errors. We have further contributed a method based on Clang to inject code changes. All of our methods surpass the current state-of-the-art of fault injection in the context of embedded systems.

4 FAULT MODELS FOR SOFTWARE FAULTS

While the previous chapter dealt with the injection of faults regarding embedded systems from a more technical point of view, the current chapter explores the underlying fault model. Having a method to inject arbitrary software faults is not enough, but one must also have a model of the faults to inject, possibly bundled with a distribution of such faults to improve the performance or accuracy of injection campaigns.

Intuitively, the first step towards a software fault model is to assess bugs which have been fixed in existing software. Chillarege et al. [12] presented the orthogonal defect classification (ODC), which is based on the idea of classifying occurred errors and will be presented in the next Section 4.1. Since this classification is too vague to be used to emulate software faults, a more fine-grained classification coming from the literature is presented in Section 4.2. Finally, Section 4.3 will discuss our contribution regarding the state-of-the-art, followed by a novel concept in Section 4.4, potential problems of our envisioned approach in Section 4.5 and potential applications of our approach in Section 4.6.

4.1 COARSE-GRAINED CLASSIFICATION OF SOFTWARE FAULTS

Chillarege et al. presented the ODC to classify faults according to a classification which allows for an unambiguous mapping [12]. First, they classify faults according to their underlying fix out of the following options. Each of the categories can be further divided into *missing* or *incorrect*.

FUNCTION Faults which affect the end-user interfaces, product interfaces, interfaces to hardware, or similar serious issues, which require a formal design change to modify the so-called requirements specification.

INTERFACE Faults in interacting with components, modules, or hardware not covered in the requirements specification.

CHECKING Faults affecting the proper validation of data before usage.

ASSIGNMENT Faults related to a small part of the code responsible for the initialization of a data structure, or similar.

TIMING/SERIALIZATION Faults involving the use of shared and real-time resources.

BUILD/PACKAGE/MERGE Faults related to library systems, management of changes, version control, and build processes in general.

DOCUMENTATION Faults regarding (internal) documentation, release of change notes or similar.

ALGORITHM Faults affecting the efficiency or proper function of a task but without the need for a formal design change request.

These categories can be associated with different stages of the software development process, where they are likely to occur. Table 4.1 shows some of

STAGE	DEFECT TYPE			
	Function	Checking	Timing	Algorithm
Design	•			
Low-Level Design			•	
Code		•		•
High-L. Design Insp.	•			
Low-L. Design Insp.			•	
Code Inspection		•		•
Unit Test		•		•
Func Test	•			•
System Test			•	

Table 4.1: Association between defect type and stages of the software development process, adapted from Chillarege et al. [12].

these associations, as presented in the original article. Each category can be associated with design phases, actual programming activities, or inspection phases. The association gives hints where faults of a specific defect type are most likely to occur. If they occur at different stages, this can be interpreted as a hint that verification stages are not properly working.

Additionally, a defect trigger is identified for each classified defect. It characterizes the circumstances which allow a fault to cause an error and finally a failure. While the defect type can be specified only when the underlying fault has been identified, the defect trigger can be specified very early when encountering a defect.

4.2 FINE-GRAINED CLASSIFICATION OF SOFTWARE FAULTS

The coarse-grained classification of ODC is not useful for emulating software faults because it is unclear how to modify software to emulate a fault. Durães et al. therefore proposed a more fine-grained classification based on the ODC classification, which is based on necessary modifications to source code to emulate a fault [17]. Table 4.2 shows a subset of the proposed fault types, although the original article lists more. Later research has shown that many types are so rare that they can be ignored [53].

We recommend to read the following article before this chapter:

> Ulrich Thomas Gabor et al. "High-Accuracy Software Fault Injection in Source Code with Clang." In: *Proceedings of the 24th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '19)*. Pasadena, CA, USA: IEEE Press, Dec. 2019, pp. 75–84. DOI: 10.1109/PRDC47002.2019.00029

In our article, we have listed multiple problems current methods for the injection of code changes suffer from:

MACROS AND COMPILE-TIME LANGUAGE CONSTRUCTS Most approaches neglect compile-time language constructs like C++ templates, although they are a Turing-complete addition to the C++ language. Even macros,

FAULT TYPE	EXPLANATION
MFC	Missing function call
MIA	Missing IF construct around statements
MIEB	Missing IF construct+statements+ELSE before statements
MIES	Missing IF construct+statements+ELSE+statements
MIFS	Missing IF construct+statements
MLAC	Missing AND clause in branch condition
MLOC	Missing OR clause in branch condition
MLPA	Missing small and localized part of the algorithm
MVAE	Missing variable assignment with an expression
MVAV	Missing variable assignment using a value
MVIV	Missing variable initialization using a value
WAEP	Wrong arithmetic expression in function call parameter
WPFV	Wrong variable used in parameter of function call
WVAV	Wrong value assigned to variable

Table 4.2: Fault types used for software fault injection in our experiments [17].

which have been around for a long time and are also practically Turing-complete, are often ignored. Some approaches operate directly on the assembly, while others operate on preprocessed code. We have demonstrated that especially the ability of macros to impact many source code positions at once can have a dramatic effect if there is a fault in a macro. A tool operating on a preprocessed or binary file should inject the fault at all macro expansions to correctly emulate the fault.

IMAGE MODIFICATIONS AND THEIR IMPACT ON FAILURES Concerning image modifications, we have shown that some approaches, especially those that add support code for executing a fault injection campaign into the injection target, can have a dramatic effect on the size of the resulting binary. In our experiments, the binary size doubled to tripled in size. Especially in the context of embedded systems, such growth may be a problem for the restricted memory of these systems. The changes can additionally have an effect on the faulty behavior of the injection target due to wild pointers or changes in execution.

RECOGNIZING CODE IN OPTIMIZED ASSEMBLY Especially approaches that operate directly on the binary, but also those operating on an intermediate representation of a compilation, suffer from the problem that structures of the source code, where faults should be injected, must first be found. Most authors seem to use unoptimized code for their evaluations, but the second optimization level `-O2` is used in practice regularly. We have shown by examples, that it is impossible to identify some injection targets in such an optimized code because the corresponding source code was optimized away. The latter can stem from basic optimizations like inlining of functions to more complex operations like loop-unrolling.

Additionally to these problems of injection methods, we have identified problems in the specification of the currently used fault model [17]. While the authors have provided extensive specification of faults in text form and even

published an appendix for their article with more details, we encountered several uncertainties when implementing our injection tool. We doubt that it is possible to compose an unambiguous specification in text form covering all possibilities, especially since programming languages and development processes evolve.

The rest of this chapter is based on our findings regarding the specification of fault models and the accuracy of injections, i. e., proximity between generated faults to inject and the faults which should be injected according to the underlying fault model [25], but will take the ideas a little bit further than what has already been published.

4.3 IMPROVING ACCURACY OF FAULT MODELS FOR FAULT INJECTION

We already questioned the accuracy of injection tools regarding the underlying fault model in Chapter 3, i. e., how close the generated injections are to those which should have been generated according to the fault model.

In this section, another form of accuracy comes into play: the accuracy of fault models in comparison to real software faults. We already published findings related to this form of accuracy [25, 26], arguing that fault types and distributions may change and that a textual description of fault types might lead to ambiguities when implementing software fault injection (SFI) tools. Therefore, we propose a solution closer to source code.

Our vision is an automatic fault classifier, which is capable of characterizing faults automatically according to some preset classification. A tool like this faces two main obstacles:

1. determining which source code modifications constitute a bug, and
2. mapping the source code modifications to an abstract model.

Luckily, software development processes have improved a lot in recent years. For example, Github[1] provides a ticket system with version control system (VCS) integration for all projects, which allows for a mapping of source code changes to a specific bug report.

Furthermore, there have been improvements regarding the determination of differences between source code, notably the so-called Gumtree algorithm [20], which computes differences between two source code files represented as graphs and works on the so-called abstract syntax tree (AST). With the introduction of Clang, source code handling of C++ code has improved in general, and also the Gumtree algorithm has been ported to Clang.

Having an automatic fault classifier would allow for the classification of many more bugs than manually. The interaction with a ticket system allows for a mapping of bugs to specific project phases. Additionally, since classification happens automatically, there exists an unambiguous decision-maker regarding the fault types. Such a decision-maker should help when implementing SFI tools, for example, because a generated fault can be classified again to check if it was correctly generated.

4.4 CONCEPT OF AN AUTOMATIC FAULT CLASSIFIER

We already envisioned the process to automatically classify faults according to a given classification, see Figure 4.1 for an overview.

1 https://github.com

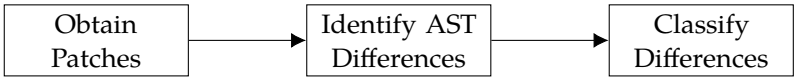

Figure 4.1: Steps of envisioned fault classifier.

Listing 4.1: Example of an edit script

```
Insert BinaryOperator: &&(10) into BinaryOperator: =(8) at 1
Insert ParenExpr(11) into BinaryOperator: &&(10) at 0
Insert BinaryOperator: ==(12) into ParenExpr(11) at 0
Insert DeclRefExpr: argc(13) into BinaryOperator: ==(12) at 0
Insert IntegerLiteral: 5(14) into BinaryOperator: ==(12) at 1
Move ParenExpr(15) into BinaryOperator: &&(10) at 1
```

The first step of this process is to obtain patches which fixed bugs. We will use Github as an example of a current hoster for the VCS Git. Github provides an application programming interface (API)[2] to obtain not only information regarding existing issues but also which Git commits are affected. It is also possible to label issues, for example with a label called "bug". Even if labels are not used for a project, it is possible to search the issue text for the occurrence of the word "bug" (or similar words) to select only issues dealing with bugs and not questions or feature requests.

The second step of our envisioned process is to gather the AST differences caused by the commits related to an issue. One standard notation is a so-called edit script, see Listing 4.1 for an example. These scripts consist of a sequence of operations that transform one AST to the other, where the operations are one of four types – insert, update, move and delete. To obtain such an edit script, first, a mapping between the two ASTs must be established from which an edit script can be built [10].

Obtaining a mapping between AST nodes can be done, for example, with the Gumtree algorithm [20], which builds a mapping in two steps similar to how a human would infer a mapping. The first step is a top-down algorithm to find the largest isomorphic subtrees, which are then mapped and called anchors. In the second step, a bottom-up algorithm tries to map nodes with a large number of common anchors, and if two nodes are matched, another algorithm will try to match the nodes of the two subtrees.

The third step is to classify edit scripts, so that each is mapped to a fault type, if this is possible. Building the classification on top of edit scripts allows to replace the mapping algorithms.

4.5 POTENTIAL PROBLEMS

It may be hard or impossible to differentiate between the obtained ticket types automatically. Modern software project management services like Github provide labels for issues. These can be set by the creator of an issue or members of the corresponding project. Using these labels thoroughly would already suffice to enable a distinction between bugfixes and other issue types. An alternative could be natural language processing (NLP).

It may be adequate or necessary to build the classification algorithm not on top of edit scripts but directly on the output of the mapping algorithm, because

2 https://developer.github.com/v3/

the classification might require more information regarding the concrete AST than the condensed information edit scripts provide.

Consider, for example, a condition in the head of a loop construct. If the change only adds tokens to the source code at a specific location, then the edit script will only contain `Insert` operations and a reference to the parent node. If this was not the outermost operator, then this parent node can be another binary operator node by which one cannot tell if the whole condition is part of a loop header or only part of a boolean assignment. One would need to traverse the parent nodes up until a statement node is reached, which gives enough context to distinguish different fault types. This information regarding parent nodes is not present in an edit script.

4.6 POTENTIAL APPLICATIONS OF THE CLASSIFIER

Implementing our idea as a tool would allow us to answer multiple research questions, and the findings could improve the current situation regarding software fault injection and especially software fault models. Consider the following list for an impression on what questions such a tool could answer:

1. Research might lead to insights into which additional information should be gathered in a ticket system to ease the automatic classification of existing bugs for modeling purposes.
2. It is unclear if the fault model set up multiple years ago is still valid for current software projects.
3. Furthermore, it is unclear if the fault types and their distribution apply to all types of software projects equally. Especially software coming from the embedded system and the operating system field might be subject to special fault types or distributions.
4. Additionally, it is unclear if fault types or the distribution of the changes during the lifetime of a software project.
5. The fault types seem too simplistic to emulate what occurs as bugs in current software projects correctly. Often a patch consists of more than one type of fault. An automatic classifier could be used to split patches and obtain statistics on which fault types regularly occur together to form one fault.

In this chapter, we highlighted potential pitfalls when specifying fault models and raised multiple questions regarding their validity, given the change of software development processes in recent years. Additionally, we demonstrated the concept of an automatic fault classifier, which could be used to answer multiple of the raised research questions and, therefore, could improve the accuracy of fault injection experiments.

5 FAULT LOCALIZATION IN DEPLOYED EMBEDDED SYSTEMS

As described in Section 2.4, there exist various approaches for fault localization. We wanted to know if it is possible to localize faults in-field, i. e., in a deployed embedded system. These systems come with additional difficulties, because often embedded systems are context-aware or must at least run in different and potentially unknown environments, but are at the same time not easily accessible.

We present two possible methods to achieve this goal. The first one is based on a program spectrum analysis, where information gathered at runtime, the so-called spectrum, is used together with information about succeeding and failing results to pinpoint potential fault locations. This idea has already been published and will be presented in Section 5.1. The second method is only presented as a concept in this thesis and uses models and assertions to verify the execution at runtime, see Section 5.2.

5.1 PROGRAM SPECTRUM-BASED FAULT LOCALIZATION

Most of the state-of-the-art approaches from literature require high computational power or much storage. They inspect all statements [36], data and control dependencies in programs in the form of a program dependence graph (PDG) [59], or are traditionally based on the results of the execution of failing or even all test cases of a test suite [73], which requires many resources by design. A few authors also focus on efficiency, for example, by using dynamic code coverage starting with more coarse instrumentation [58].

Regularly, high resource consumption is no problem on desktop computers, but it is not adequate to pair an embedded system with a desktop computer in-field. For an analysis at runtime, we came up with the idea to pair two embedded systems, where one is the system-under-test, and the other one is tasked to monitor the system-under-test.

The idea of using two small systems comes with multiple advantages compared to other solutions. It is cheap, because often embedded systems are made out of small commodity hardware, and it is possible to leave the system in-field without the need for desktop computers or continuous access to the system-under-test. Having a separate system monitoring events helps to maintain a flawless report in case of deviating behavior. A separate system is more isolated and, therefore, safer against error propagation. If isolation can be achieved by other means, e. g., virtualization, both functions can be deployed on one piece of hardware as well.

As most of the presented approaches in Section 2.4 require high computational power or storage, we decided to assess spectrum-based fault localization (SBFL) in combination with models as oracles to decide when runs were successful or failing. We recommend reading our article with the published results at this point.

Ulrich Thomas Gabor et al. "Spectrum-Based Fault Localization in Deployed Embedded Systems with Driver Interaction Models." In: *Proceedings of the 38rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '19)*. Ed. by Alexander Romanovsky et al. Turku, Finland: Springer International Publishing, 2019, pp. 97–112. ISBN: 978-3-030-26601-1. DOI: 10.1007/978-3-030-26601-1_7

In summary, we assessed failure-detection oracles, transaction detectors, and suspiciousness metrics in the context of embedded systems, i. e., a simulated combustion engine. We compared two failure-detection oracles: the software behavior graph (SBG) [44], which is a graph representation of the call hierarchy of functions, and the driver interaction model (DIM) [23], our contribution, which is an automaton modeling the interaction with periphery. We split the execution in slices based on time and used each slice as a separate transaction, as done by others [1], and we used multiple available suspiciousness metrics from literature [73]. For the generation of spectra, we viewed each called function as component and also compared it to method call sequence hit spectra (MCSHS) [14], an approach again taken from literature.

Both failure-detection oracles are models of correct software behavior. These models can either come from the specification phase, can be learned from correct software, or be developed interactively during the implementation phase. Some origins might require special care, and the use cases of our approach depend on the specific setting. Further details can be found in the corresponding publication [23].

Our evaluation revealed that using MCSHS regularly generates so much data that it cannot be assessed in real-time. Therefore, we stuck to using only the currently executed function as a component in the spectrum. In comparison, our contribution – the DIM – detected around 50 %, whereas the SBG only detected 0.05 % of the failures. Evaluating multiple available configurations with the well-known EXAM metric [72] showed that using the time-based transaction detector works quite well, i. e., in 80 % of our experiments, the first reported location to the developer to search for the fault was indeed the location of the fault.

Concluding, we have shown that our approach works in the context of embedded systems, that our monitoring approach can keep up with the generation of data by the system-under-test in real-time and that it often provides the exact location of the fault as the first result. Additionally, we have shown that our oracle type DIM can detect failures better in comparison to the SBG.

5.2 ASSERTION-BASED FAULT LOCALIZATION

While the first method already utilized models to decide when a run was succeeding or failing, the gathered runtime information and its evaluation are used to give only a probabilistic location of the underlying fault. While this works well in some cases, it might not yield good results in cases of more complex software and interaction, where execution is not that repetitive.

The alternative presented as a concept in the current section requires predefined software models – in contrast to the first approach where corresponding models could be learned. Having a thorough software model at hand allows us to construct assertions, which check invariants at runtime. This, in turn, allows us to give more pinpointed locations if a failure is detected. The underlying idea is to automatically generate the assertions that check that the execution happens according to modeling rules.

Listing 5.1: Example OCL rules.

```
context Company
    inv: self.manager->size() = 1

context List:items : Collection(Items)
    init: Collection{}

context List::removeElement(d:Data)
    pre: oclIsInState(notEmpty)
    post: size@pre = 1 implies oclIsInState(empty)
```

The basics for this approach were developed by Meyer when developing the Eiffel programming language and the design-by-contract approach [48, 49]. When applying this approach, software designers define formal, precise, and verifiable interface specifications. Regularly, this involves information regarding the allowed input values, return values, invariants, pre- and post-conditions. Such information can be transformed into assertions so that they can be checked at runtime.

5.2.1 *Transformation of Model Constraints to Assertions*

The de facto standard for software modeling is the Unified Modeling Language (UML), which comes with the Object Constraint Language (OCL). OCL is a declarative language to annotate rules in UML models, especially invariants, pre- and postconditions [56]. It was initially developed by IBM and is now standardized by the Object Management Group (OMG). In contrast to specification documentation, which may be semi-formal text, OCL is a precise language underpinned by a formal description of its syntax. It is, therefore, possible to write parsers for it.

Multiple examples of OCL rules are shown in Listing 5.1. The first rule specifies the invariant that for each instantiation of a class `Company`, exactly one manager must be associated. The second rule specifies that, given a `List` implementation with the list items saved in a variable `items`, the variable `items` should be empty at the point of creation of an object of that class. The third rule specifies pre- and postconditions for the execution of the method `removeElement` of the list implementation. It uses so-called predefined properties, here `oclIsInState()`, to make sure that the object is in the appropriate state specified by a supporting UML state-chart before and after the execution of the method.

These examples already show many of the features OCL provides, although there are more, which we do not present for brevity. The question is how such specifications can be used to generate code, which checks the rules at runtime.

Aspect-oriented programming (AOP) can be used to augment every method execution and surround the execution with extra code, which can be used to check invariants as well as pre- and postconditions. Other programming languages like C# provide dedicated design-by-concept language constructs. While such a language feature has been proposed for addition to the C++ standard multiple times already, it has also been rejected multiple times and is now planned for C++23. Until then, AOP provides a good alternative, and even when first-class language constructs are available, AOP might provide additional benefit, as we will discuss later in this section.

Listing 5.2: Example transformation of the OCL rules given in Listing 5.1 to AspectC++.

```
aspect OCL_Company {
    advice execution("% Company::%(...)") : around() {
        assert(manager->size() = 1);
        *tjp->proceed();
        assert(manager->size() = 1);
    }
};

aspect OCL_List {
    advice construction("List") : after() {
        assert(items->empty());
    }

    advice execution() : around() {
        oclIsInState(notEmpty);
        *tjp->proceed();
        oclIsInState(empty);
    }
};
```

A devised transformation of the OCL examples of Listing 5.1 to AspectC++ aspects can be seen in Listing 5.2. While the example code requires some implementations which currently do not exist, the straightforward transformation of OCL rules to AspectC++ should become clear. One missing implementation is, for example, that of `oclIsInState`, which requires that the state of the object is traced somewhere.

The idea to transform OCL rules to AOP language constructs is not new [18], but we do not know of anyone who has thought about doing this for AspectC++. A general approach to transform OCL rules to any implementation language was presented by Moiseev et al., which generates pseudo-code fitting the structural similarities of a class of programming languages and then converts this into the target language [51]. They evaluate their approach using four languages: Java, Python, Haskell, and O'Haskell. Furthermore, they claim that it should be possible to add C++ to this list. However, they do not discuss how assertions generated by their approach can be integrated into a software project or how changes to the OCL rules can be propagated into already augmented source code.

Other approaches make use of source-to-source transformation, thereby allowing to keep code written by developers and automatically generated assertions separated and allowing to redo the process, when OCL rules change. One approach uses C++ templates [65]; another one is facilitating OpenC++ [11]. None of these solutions is still maintained today, and since OCL changed substantially between versions 1 and 2 and these approaches were all presented before the release of version 2, they are – in general – rendered unusable for current software models. Therefore, only the concepts of these approaches remain today. In contrast, AOP is a multi-purpose source-to-source transformation approach, and methods using it are not prone to be abandoned like special-purpose tools.

Willink has identified multiple problems when transforming current OCL rules given in version 2 to Java, e. g., regarding the problem of implementing OCL's unlimited numbers [71]. To the best of our knowledge, this is the most

current overview of limitations when transforming modern OCL rules to implementation languages. Since the standard libraries of Java and C++ are similarly powerful, similar limitations will apply when transforming modern rules to C++.

Concluding, despite all these alternatives, we were not able to find any maintained software to extend modern C++ source code with assertions coming from OCL constraints given in an up-to-date version of OCL. A transformation of OCL rules to AspectC++ code seems to be the most reasonable approach currently, because it would not require special-purpose implementations. Instead, most requirements can be easily fulfilled because aspects have access to all C++ features and all methods and attributes of the classes. Nevertheless, transforming all the state changing rules and all OCL features to C++ code remains an open challenge. Solving this challenge requires not only an OCL parser but also the implementation of various data structures/algorithms to implement all OCL features and a transformation of a state-chart to a corresponding state tracer. However, we do not expect significant problems implementing this with AspectC++ in contrast to pure C++.

5.2.2 *Applicability of the Approach*

If an implementation of a transformer from OCL to AspectC++ existed, it would allow for the inline generation of checking code, which could check for deviating behavior at runtime. As soon as such a deviation was recognized, the system could notify a second embedded system, which only keeps track of occurred errors and the rule which discovered the problem. A developer could use this information to pinpoint the underlying fault. A second isolated system for monitoring improves the reliability of the monitoring system, as the system-under-test might behave unexpectedly in case of an error. If other isolation mechanisms are available, it might be possible to combine both functions on the same hardware.

It remains unclear if our proposed approach is feasible. Dzidek found that when transforming OCL rules to AspectJ for Java applications, the size of the binary tripled in their example, and the execution time increased between 50 % and 100 %, while the memory footprint increased unsubstantially [18]. Although this seems high, the overhead varies and is mainly affected by the number of transformed OCL rules, i. e., more rules result in more code and more execution time. It can be expected that the overhead of an implementation using AspectC++ will be similar.

For embedded systems, it is likely not possible to activate all assertions at the same time. Checking assertions can have a significant effect on the runtime, and converting all OCL rules to assertions can result in binaries that are too large for the embedded device. Therefore, we propose an approach which sorts classes topologically in a tree structure (ignoring circular dependencies for now) and activates assertions only on the upper layers. Each time a failure is detected, one assertion should have been violated. Further search for the fault then can concentrate on that subtree of classes and allows to activate corresponding assertions on lower levels. This approach can be repeated until the assertion has been found that cannot be unfolded further. Depending on if it was possible to integrate all assertions at once, this search process could either be orchestrated by the monitoring system automatically, or it might require new compilations and, therefore, likely requires human interaction.

Another problem might be that a failing system might not be capable anymore of verifying assertions or logging violations. A possible solution to this

problem might be that assertions are checked on another system, which requires the transmission of data to verify them.

In summary, we presented two approaches in this chapter to localize faults, especially in deployed embedded systems. Both approaches make use of two isolated functional units, one being the system-under-test and the other one being a monitor, which can be deployed on two separate embedded systems or hardware providing other appropriate isolation mechanisms. The first approach uses aggregated execution information to give a probabilistic location of the root cause. The second approach uses assertions generated from a software model to give an exact location of the deviating behavior.

6 CONCLUSION AND OUTLOOK

This thesis by publication presented our contributions to the current state-of-the-art regarding fault injection and localization in embedded systems. First, we have shown how useful aspect-oriented programming (AOP) is for the three classes of software fault injection (SFI), especially in the context of embedded systems, which have additional limitations in comparison to a desktop computer or server. Then, a thorough assessment of the currently used fault models revealed multiple problems. While we have solved some problems, we left others as questions for future research and envisioned concepts for this research. Additionally, contrasting the injection of faults, we have also examined how faults in deployed embedded systems can be localized by using program spectrum or assertions.

In detail, we have contributed a fault injection procedure for interface injection based on AspectC++ and have demonstrated that AOP, in general, is especially useful for this application. Additionally, we assessed how useful AOP is for the injection on the other abstraction layers and concluded that other approaches are more promising. While contributing a method to inject faults into source code based on Clang, we assessed the state-of-the-art fault models and found multiple limitations. We concluded that an automatic fault classifier would allow us to answer various research questions brought up while working on this thesis and described our concept of such a classifier.

Furthermore, while spectrum-based fault localization techniques are not new, it was unknown if it is possible to implement such algorithms on deployed embedded systems. We have shown that these algorithms can run on embedded systems and are also useful to detect faults in deployed embedded systems by using automatic oracles, i. e., automata, which are a model of correct interaction with the periphery. Additionally, we depicted a concept on how constraints given in a software model, i. e., Object Constraint Language (OCL) constraints, can be transformed into assertions by using AspectC++, which enables non-probabilistic fault localization techniques in deployed embedded systems.

6.1 OPEN QUESTIONS

We identified multiple open questions throughout this thesis that leave room for future work. Although most of them were already mentioned throughout this thesis, the current section will give a focused overview.

6.1.1 *SFI for Embedded Systems*

Most of the open questions brought up in this thesis regarding SFI are related to the injection of code changes into C++ source code.

1. While we have shown that ignoring C++ peculiarities can have drastic effects on experimental results, it is unclear how these effects manifest for real software projects. Future work should evaluate the differences between, on the one hand, experiments ignoring compile-time constructs, e. g., macros and shared headers, and, on the other hand, experiments,

which perform injections as a human would introduce faults on the source-code level.

2. Depending on results regarding the first problem, it might be necessary to assess again how close injections into binary/intermediate code are to injections into source code. If even the negligence of macros turns out to be problematic, then injections into binary/intermediate code should be problematic as well.

6.1.2 *Fault Models for Software Faults*

The implementation of a fault injection tool on the source code level for embedded systems required an assessment of the fault model and its description, which in turn led to multiple questions mainly regarding the adequacy of precious models to modern software engineering:

1. It is unclear if the currently used fault types are still relevant given developments of software engineering techniques in the last years.

2. The currently used fault types are quite basic. From our experience with fixing bugs, a patch often consists of multiple fault types, which are fixed simultaneously. An in-depth assessment of performed bug fixes should show if current bugs can be simulated given single fault types or if combinations of fault types are required.

3. Furthermore, it is unclear if the currently used distribution of fault types is still correct, and if the distribution can be applied to every software project in every domain. Different domains, e. g., operating system and application software, might pose quite different distributions of fault types.

6.1.3 *Fault Localization in Deployed Embedded Systems*

As presented, our work enriched the current state-of-the-art regarding fault localization by assessing current techniques in the context of deployed embedded systems [23]. Since the assessment showed room for improvement, we contributed a new type of oracle, the driver interaction model (DIM), which is especially useful for embedded systems, and discussed assertion-based fault localization in this thesis. While our results were promising, we also identified possible future work:

1. The presented DIM, which defines the interaction of drivers with the underlying hardware, worked already well regarding our evaluation. Nevertheless, it might be possible to extend the representativeness, for example, by using extended probabilistic automata [19] instead.

2. Other techniques can be used as oracles too. For example, the OCL-based assertions presented in Section 5.2 can be used in combination with spectrum-based fault localization (SBFL). It is still unclear if the use of other oracles might improve the localization accuracy, i. e., the EXAM score.

Considering the variety of open research questions, it becomes evident that the research area of fault injection and localization, especially in the domain of embedded systems, still offers room for future research efforts.

BIBLIOGRAPHY

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

[2] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[3] James H. Barton, Edward W. Czeck, Zary Z. Segall, and Daniel P. Siewiorek. "Fault Injection Experiments Using FIAT." In: *IEEE Transactions on Computers* 39.4 (Apr. 1990), pp. 575–582. ISSN: 0018-9340. DOI: 10.1109/12.54853.

[4] Tania Basso, Regina L. O. Moraes, Bruno P. Sanches, and Mario Jino. "An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults." In: *Workshop de Testes e Tolerância a Falhas* (2009), pp. 150–155. ISSN: 2325-6648. DOI: 10.1109/DSNW.2010.5542602.

[5] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. "CiAO/IP: A Highly Configurable Aspect-Oriented IP Stack." In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (*MobiSys '12*) (Low Wood Bay, Lake District, UK). New York, NY, USA: ACM Press, June 2012, pp. 435–448. ISBN: 978-1-4503-1301-8. DOI: 10.1145/2307636.2307676.

[6] Christoph Borchert and Olaf Spinczyk. "Hardening an L4 Microkernel Against Soft Errors by Aspect-Oriented Programming and Whole-Program Analysis." In: *ACM Operating Systems Review* 49.2 (Jan. 2016), pp. 37–43. ISSN: 0163-5980. DOI: 10.1145/2883591.2883600.

[7] Pete Broadwell, Naveen Sastry, and Jonathan Traupman. "FIG: A Prototype Tool for Online Verification of Recovery." In: *In Workshop on Self-Healing, Adaptive and Self-Managed Systems*. 2002.

[8] Fraser Brown, Andres Nötzli, and Dawson Engler. "How to Build Static Checking Systems Using Orders of Magnitude Less Code." In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA). ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 143–157. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872364.

[9] Maarten Bynens, Eddy Truyen, and Wouter Joosen. "A System of Patterns for Reusable Aspect Libraries." In: *Transactions on Aspect-Oriented Software Development VIII*. Ed. by Shmuel Katz, Mira Mezini, Christine Schwanninger, and Wouter Joosen. LNCS 6580. Berlin, Heidelberg: Springer, 2011, pp. 46–107. ISBN: 978-3-642-22031-9. DOI: 10.1007/978-3-642-22031-9_2.

[10] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. "Change Detection in Hierarchically Structured Information." In: *SIGMOD Rec.* 25.2 (June 1996), pp. 493–504. ISSN: 0163-5808. DOI: 10.1145/235968.233366.

[11] Shigeru Chiba and Takashi Masuda. "Designing an Extensible Distributed Language with a Meta-Level Architecture." In: *ECOOP' 93 — Object-Oriented Programming*. Ed. by Oscar M. Nierstrasz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 482–501. ISBN: 978-3-540-47910-9. DOI: 10.1007/3-540-47910-4_24.

[12] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. "Orthogonal Defect Classification – A Concept for In-Process Measurements." In: *IEEE Transactions on Software Engineering* 18.11 (Nov. 1992), pp. 943–956. ISSN: 0098-5589. DOI: 10.1109/32.177364.

[13] Domenico Cotroneo, Anna Lanzaro, Roberto Natella, and Ricardo Barbosa. "Experimental Analysis of Binary-Level Software Fault Injection in Complex Software." In: *Ninth European Dependable Computing Conference (EDCC)*. May 2012, pp. 162–172. DOI: 10.1109/EDCC.2012.12.

[14] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. "Lightweight Defect Localization for Java." In: *ECOOP 2005 - Object-Oriented Programming: 19th European Conference, Glasgow, UK, July 25-29, 2005. Proceedings*. Ed. by Andrew P. Black. Berlin, Heidelberg: Springer, 2005, pp. 528–550. ISBN: 978-3-540-31725-8. DOI: 10.1007/11531142_23.

[15] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Juan José Domínguez-Jiménez, Antonio García-Domínguez, and Francisco Palomo-Lozano. "Class mutation operators for C++ object-oriented systems." In: *annals of telecommunications - annales des télécommunications* 70.3 (2015), pp. 137–148. ISSN: 1958-9395. DOI: 10.1007/s12243-014-0445-4.

[16] Jessica Díaz, Jennifer Pérez, Carlos Fernández-Sánchez, and Juan Garbajosa. "Model-to-Code Transformation from Product-Line Architecture Models to AspectJ." In: *Proceedings of the 2013 39th Euromicro Conference on Software Engineering and Advanced Applications*. SEAA '13. USA: IEEE Computer Society, 2013, pp. 98–105. ISBN: 9780769550916. DOI: 10.1109/SEAA.2013.11.

[17] João A. Durães and Henrique S. Madeira. "Emulation of Software Faults: A Field Data Study and a Practical Approach." In: *IEEE Transactions on Software Engineering* 32.11 (Nov. 2006), pp. 849–867. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.113.

[18] Wojciech Dzidek. "Using Aspect-Oriented Programming to Instrument OCL Contracts in Java." PhD thesis. Carleton University Research Virtual Environment, 2004.

[19] S. S. Emam and J. Miller. "Inferring Extended Probabilistic Finite-State Automaton Models from Software Executions." In: *ACM Trans. Softw. Eng. Methodol.* 27.1 (June 2018), 4:1–4:39. ISSN: 1049-331X. DOI: 10.1145/3196883.

[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. "Fine-grained and Accurate Source Code Differencing." In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Vasteras, Sweden: ACM, 2014, pp. 313–324. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642982.

[21] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-oriented Software Development*. 1st. Boston, MA, USA: Addison-Wesley, Oct. 2004. ISBN: 978-0321219763.

[22] Daniel Friesel, Markus Buschhoff, and Olaf Spinczyk. "Annotations in Operating Systems with Custom AspectC++ Attributes." In: *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (*PLOS '17*). PLOS'17. Shanghai, China: ACM, 2017, pp. 36–42. ISBN: 978-1-4503-5153-9. DOI: 10.1145/3144555.3144561.

[23] Ulrich Thomas Gabor, Simon Dierl, and Olaf Spinczyk. "Spectrum-Based Fault Localization in Deployed Embedded Systems with Driver Interaction Models." In: *Proceedings of the 38rd International Conference on Computer Safety, Reliability and Security* (*SAFECOMP '19*). Ed. by Alexander Romanovsky, Elena Troubitsyna, and Friedemann Bitsch. Turku, Finland: Springer International Publishing, 2019, pp. 97–112. ISBN: 978-3-030-26601-1. DOI: 10.1007/978-3-030-26601-1_7.

[24] Ulrich Thomas Gabor, Christoph-Cordt von Egidy, and Olaf Spinczyk. "Interface Injection with AspectC++ in Embedded Systems." In: *Proceedings of the 19th IEEE International Symposium on High Assurance Systems Engineering* (*HASE '19*). IEEE Press, Jan. 2019, pp. 131–138. DOI: 10.1109/HASE.2019.00028.

[25] Ulrich Thomas Gabor, Daniel Siegert, and Olaf Spinczyk. "High-Accuracy Software Fault Injection in Source Code with Clang." In: *Proceedings of the 24th IEEE Pacific Rim International Symposium on Dependable Computing* (*PRDC '19*). Pasadena, CA, USA: IEEE Press, Dec. 2019, pp. 75–84. DOI: 10.1109/PRDC47002.2019.00029.

[26] Ulrich Thomas Gabor, Daniel Siegert, and Olaf Spinczyk. "Software-Fault Injection in Source Code with Clang." In: *Proceedings of the 32th International Conference on Architecture of Computing Systems* (*ARCS '19*), *Workshop Proceedings*. 2019, pp. 1–6. ISBN: 978-3-8007-4957-7.

[27] Luca Gazzola, Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. "An Exploratory Study of Field Failures." In: *2017 IEEE 28th International Symposium on Software Reliability Engineering* (*ISSRE*). Oct. 2017, pp. 67–77. DOI: 10.1109/ISSRE.2017.10.

[28] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. "EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments." In: *2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing*. Dec. 2013, pp. 3–140. DOI: 10.1109/PRDC.2013.12.

[29] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Mutations: How close are they to real faults?" In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. Nov. 2014, pp. 189–200. DOI: 10.1109/ISSRE.2014.40.

[30] Michael Grottke, Dong Seong Kim, Rajesh Mansharamani, Manoj Nambiar, Roberto Natella, and Kishor S. Trivedi. "Recovery From Software Failures Caused by Mandelbugs." In: *IEEE Transactions on Reliability* 65.1 (Mar. 2016), pp. 70–87. ISSN: 0018-9529. DOI: 10.1109/TR.2015.2452933.

[31] Michael Grottke and Kishor S. Trivedi. "Software Faults, Software Aging and Software Rejuvenation." In: *The Journal of Reliability Engineering Association of Japan* 27.7 (Oct. 2005), pp. 425–438. ISSN: 09192697. DOI: 10.11348/reajshinrai.27.7_425.

[32] Stefan Hanenberg and Stefan Endrikat. "Aspect-orientation is a rewarding investment into future code changes – As long as the aspects hardly change." In: *Information and Software Technology* 55.4 (2013), pp. 722–740. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2012.09.005.

[33] Lena Herscheid, Daniel Richter, and Andreas Polze. "Hovac: A Configurable Fault Injection Framework for Benchmarking the Dependability of C/C++ Applications." In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. Aug. 2015, pp. 1–10. DOI: 10.1109/QRS.2015.12.

[34] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. "Fault Injection Techniques and Tools." In: *Computer* 30.4 (Apr. 1997), pp. 75–82. ISSN: 0018-9162. DOI: 10.1109/2.585157.

[35] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 672–681. ISBN: 978-1-4673-3076-3. DOI: 10.1109/ICSE.2013.6606613.

[36] James A. Jones and Mary Jean Harrold. "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique." In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: ACM, 2005, pp. 273–282. ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101949.

[37] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. "FERRARI: A Flexible Software-Based Fault and Error Injection System." In: *IEEE Transactions on Computers* 44.2 (Feb. 1995), pp. 248–260. ISSN: 0018-9340. DOI: 10.1109/12.364536.

[38] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An Overview of AspectJ." In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Ed. by J. Lindskov Knudsen. Vol. 2072. Lecture Notes in Computer Science. Berlin, Germany: Springer, June 2001, pp. 327–354. DOI: 10.1007/3-540-45337-7_18.

[39] Nobuo Kikuchi, Takeshi Yoshimura, Ryo Sakuma, and Kenji Kono. "Do Injected Faults Cause Real Failures? A Case Study of Linux." In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. Nov. 2014, pp. 174–179. DOI: 10.1109/ISSREW.2014.104.

[40] Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. "Evaluating Distortion in Fault Injection Experiments." In: *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*. Jan. 2014, pp. 25–32. DOI: 10.1109/HASE.2014.13.

[41] Erik van der Kouwe and Andrew Tanenbaum. "HSFI: Accurate Fault Injection Scalable to Large Code Bases." In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2016, pp. 144–155. DOI: 10.1109/DSN.2016.22.

[42] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem." In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176.

[43] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–88. ISBN: 0-7695-2102-9. DOI: 10.1109/CGO.2004.1281665.

[44] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S. Yu. "Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs." In: *Proceedings of the 2005 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 2005, pp. 286–297. ISBN: 978-0-89871-593-4. DOI: 10.1137/1.9781611972757.26.

[45] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. "Generic Advice: On the Combination of AOP with Generative Programming in AspectC++." In: *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering* (*GPCE '04*). Ed. by G. Karsai and E. Visser. Vol. 3286. Lecture Notes in Computer Science. Berlin, Germany: Springer, Oct. 2004, pp. 55–74. DOI: 10.1007/978-3-540-30175-2_4.

[46] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. "On the Configuration of Non-Functional Properties in Operating System Product Lines." In: *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (*AOSD-ACP4IS '05*). Chicago, IL, USA: Northeastern University, Boston (NU-CCIS-05-03), Mar. 2005, pp. 19–25.

[47] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. "Lean and Efficient System Software Product Lines: Where Aspects Beat Objects." In: *Transactions on AOSD II*. Ed. by Awais Rashid and Mehmet Akşit. Lecture Notes in Computer Science 4242. Springer, 2006, pp. 227–255. DOI: 10.1007/11922827_8.

[48] Bertrand Meyer. "Eiffel: A Language and Environment for Software Engineering." In: *Journal of Systems and Software* 8.3 (1988), pp. 199–246. ISSN: 0164-1212. DOI: 10.1016/0164-1212(88)90022-2.

[49] Bertrand Meyer. "Applying 'Design by Contract'." In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 1558-0814. DOI: 10.1109/2.161279.

[50] MIRA Limited. *MISRA C:2012*. Nuneaton, UK: MIRA Limited, 2013.

[51] Rodion Moiseev, Shinpei Hayashi, and Motoshi Saeki. "Generating Assertion Code from OCL: A Transformational Approach Based on Similarities of Implementation Languages." In: *Model Driven Engineering Languages and Systems*. Ed. by Andy Schürr and Bran Selic. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 650–664. ISBN: 978-3-642-04425-0. DOI: 10.1007/978-3-642-04425-0_52.

[52] Regina L. O. Moraes, Ricardo Barbosa, João A. Durães, Nathan Mendes, Eliane Martins, and Henrique S. Madeira. "Injection of faults at component interfaces and inside the component code: are they equivalent?" In: *2006 Sixth European Dependable Computing Conference*. Oct. 2006, pp. 53–64. DOI: 10.1109/EDCC.2006.16.

[53] Roberto Natella, Domenico Cotroneo, Joao A. Duraes, and Henrique S. Madeira. "On Fault Representativeness of Software Fault Injection." In: *IEEE Transactions on Software Engineering* 39.1 (Jan. 2013), pp. 80–96. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.124.

[54] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. "Assessing Dependability with Software Fault Injection: A Survey." In: *ACM Comput. Surv.* 48.3 (Feb. 2016), 44:1–44:55. ISSN: 0360-0300. DOI: 10.1145/2841425.

[55] Nicholas Nethercote and Julian Seward. "Valgrind: A Program Supervision Framework." In: *Electronic Notes in Theoretical Computer Science* 89.2 (2003). RV '2003, Run-time Verification (Satellite Workshop of CAV '03), pp. 44–66. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)81042-9.

[56] OMG. *OMG Object Constraint Language (OCL), Version 2.4*. Object Management Group, Feb. 2014. URL: http://www.omg.org/spec/OCL/2.4/.

[57] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. "What is Keeping My Phone Awake?: Characterizing and Detecting No-sleep Energy Bugs in Smartphone Apps." In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. MobiSys '12. Low Wood Bay, Lake District, UK: ACM, 2012, pp. 267–280. ISBN: 978-1-4503-1301-8. DOI: 10.1145/2307636.2307661.

[58] Alexandre Perez, Rui Abreu, and André Riboira. "A dynamic code coverage approach to maximize fault localization efficiency." In: *Journal of Systems and Software* 90 (2014), pp. 18–28. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.12.036.

[59] Manos Renieres and Steven P. Reiss. "Fault Localization With Nearest Neighbor Queries." In: *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* Oct. 2003, pp. 30–39. DOI: 10.1109/ASE.2003.1240292.

[60] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems." In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. ISSN: 00029947. DOI: 10.2307/1990888.

[61] Bruno Pacheco Sanches, Tânia Basso, and Regina Moraes. "J-SWFIT: A Java Software Fault Injection Tool." In: *5th Latin-American Symposium on Dependable Computing (LADC)*. Apr. 2011, pp. 106–115. DOI: 10.1109/LADC.2011.20.

[62] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "PhASAR: An Inter-procedural Static Analysis Framework for C/C++." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 393–410. ISBN: 978-3-030-17465-1. DOI: 10.1007/978-3-030-17465-1_22.

[63] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." In: *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 309–318. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[64] Olaf Spinczyk and Daniel Lohmann. "The Design and Implementation of AspectC++." In: *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software* 20.7 (Oct. 2007), pp. 636–651. DOI: 10.1016/j.knosys.2007.05.004.

[65] Kurt Stirewalt and Spencer Rugaber. "Automated Invariant Maintenance Via OCL Compilation." In: *Model Driven Engineering Languages and Systems*. Ed. by Lionel Briand and Clay Williams. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 616–632. ISBN: 978-3-540-32057-9. DOI: 10.1007/11557432_46.

[66] Yulei Sui and Jingling Xue. "SVF: Interprocedural Static Value-flow Analysis in LLVM." In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: ACM, 2016, pp. 265–266. ISBN: 978-1-4503-4241-4. DOI: 10.1145/2892208.2892235.

[67] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. "Information Hiding Interfaces for Aspect-Oriented Design." In: *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Lisbon, Portugal: ACM Press, 2005, pp. 166–175. ISBN: 1-59593-014-0. DOI: 10.1145/1081706.1081734.

[68] Martin Süßkraut and Christof Fetzer. "Automatically Finding and Patching Bad Error Handling." In: *2006 Sixth European Dependable Computing Conference*. Oct. 2006, pp. 13–22. DOI: 10.1109/EDCC.2006.3.

[69] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. Tech. rep. 2003.

[70] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (California, USA). Reston, VA, USA: Internet Society, 2000, pp. 1–15.

[71] E. D. Willink. "An Extensible OCL Virtual Machine and Code Generator." In: *Proceedings of the 12th Workshop on OCL and Textual Modelling*. OCL '12. Innsbruck, Austria: ACM, 2012, pp. 13–18. ISBN: 978-1-4503-1799-3. DOI: 10.1145/2428516.2428519.

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

[73] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. "A Survey on Software Fault Localization." In: *IEEE Transactions on Software Engineering* 42.8 (Aug. 2016), pp. 707–740. ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2521368.

[74] Qiang Zeng, Mingyi Zhao, and Peng Liu. "HeapTherapy: An Efficient End-to-End Solution against Heap Buffer Overflows." In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. June 2015, pp. 485–496. DOI: 10.1109/DSN.2015.54.