Chapter 4

Software Fault Injection: A Practical Perspective

Lena Feinbube, Lukas Pirl and Andreas Polze

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.70427

Abstract

Software fault injection (SFI) is an acknowledged method for assessing the dependability of software systems. After reviewing the state-of-the-art of SFI, we address the challenge of integrating it deeper into software development practice. We present a well-defined development methodology incorporating SFI—fault injection driven development (FIDD)—which begins by systematically constructing a dependability and failure cause model, from which relevant injection techniques, points, and campaigns are derived. We discuss possibilities and challenges for the end-to-end automation of such campaigns. The suggested approach can substantially improve the accessibility of dependability assessment in everyday software engineering practice.

Keywords: fault injection, dependability, fault tolerance, testing, test-driven development

1. Introduction

On 22 October 2012, a major service degradation at Amazon Web Services (AWS)[1] affected several popular online services for several hours. It was caused by a latent memory leak bug, activated under stress due to a failed domain name system (DNS) update after a hardware maintenance event. The leaky software agent repeatedly tried in vain to contact the replaced server. In this process, memory was leaked until customer requests could no longer be handled.

AWS is a system so complex that it challenges exhaustive formal verification. The issue could, however, have been anticipated by structured experimental dependability assessment, e.g., using fault injection. Indeed, Netflix customers remained unaffected.[2] Resiliency testing had prepared the company for such events, and failover of Netflix servers worked quickly.

[1]http://aws.amazon.com/de/message/680342/
[2]http://techblog.netflix.com/2012/10/post-mortem-of-october-222012-aws.html

This incident highlights a central issue with the current state of dependability research. There is a vast gap between the impressive theoretical achievements made in past decades, including various formal methods, and real-world software engineering practice.

Similar incidents suggest that efforts to increase systems' dependability should be shifted toward the software layers. As complexity grows, these concerns are becoming ever more challenging.

2. Fundamentals

Software dependability research is as old as the science of computing itself. In the early nineteenth century, Charles Babbage designed his famous difference engines with the main intent of reducing the error rate in complex mathematical computations [1].

As we have started to rely on software in many safety-critical aspects of our daily lives, software dependability is more relevant than ever. At the same time, the problem is getting harder. If the number of flaws (bugs) per source code lines is roughly constant, software projects increasing in code size would become more and more prone to failures. This is aggravated by the complexity caused by the interaction of different components.

Therefore, software dependability needs to take a holistic, system-wide approach, and dependability means need to scale to increasingly complex software projects. This includes means for fault forecasting and dependability assessment, which are the focus of this chapter.

2.1. Terminology

Software dependability has been characterized by a broad range of terminology [2]. We employ the wording by Avižienis, Kanoun, Kopetz, Landwehr, Laprie, and Randell, described in multiple documents dated between 1985 and 2004. The most comprehensive version [3] is used as main reference. In this terminology, dependability is threatened first by *faults*:

> *A fault is the adjudged or hypothesized cause of an error.*

In software, the terms *bug*, *defect*, or *flaw* are often used as synonyms for *fault*. When activated, a fault can lead to an undesired system state, denoted as *error*:

> *An error is that part of the system state that may cause a subsequent failure.*

There is an n:m relationship between faults and their resulting error states. A fault, when activated under varying environmental and internal conditions, might lead to different error states. In turn, each error state might be caused by different—potentially multiple—faults. Finally, error states may lead to an externally visible *failure*:

> *A system failure is an event that occurs when the delivered service deviates from correct service.*

2.2. Dependability assessment

How can the dependability of a complex software system be assessed and compared with other versions or other systems?

The complexity of software is caused by three factors:

1. The internal state space of the program, given by all variables and their values.
2. The space of input and output values of the program, which may be infinite.
3. Interaction with the environment, which is influenced, for instance, by scheduling, resource states, and interfaces to other software components.

These complexities challenge the scalability of any dependability assessment approach. There are two ends to the spectrum of such methodologies, which we classify as *formal methods* and *empirical methods*. While the spectrum is continuous and the boundaries may be fuzzy, the main differences lie in their coverage, in the assets they use (static source code vs. runtime information), and in the scalability to large software systems.

The aim of formal methods is to exhaustively prove that a program (an implementation) obeys a specification. Various efforts toward complete formal software verification have been made despite theoretical limitations as well as scalability issues.

Empirical methods for software dependability take the approach of showing that a software system operates correctly at the example of one or many executions of the program. While such approaches never provide absolute guarantees, they are generally better scalable and applicable to larger, more complex, and rapidly evolving systems. Of course, as the often-cited Dijkstra famously noted,[3]

Program testing can be used to show the presence of bugs, but never to show their absence!

2.3. Software fault injection

Software fault injection (*SFI*) denotes the artificial insertion—*injection*—of faults and error states into a running software system. It can be applied beyond the limits of formal verification methods because it does not assume a complete formal specification of the system under test. The experimental approach of SFI can be implemented efficiently and with little intrusiveness.

In the simplest case, SFI can confront an interface with randomly generated values. More sophisticated SFI operates based on detailed failure cause models and can rely on formal specifications to work more efficiently and to guarantee certain fault space coverage criteria. SFI tools can compare the dependability of different systems and answer research questions such as:

- How high is the coverage of fault tolerance mechanisms and how well do they work?
- How do faults influence quality metrics such as performance, precision, or availability?
- Are there single points of failure in the system?

Fault injection can be considered a complimentary technique to software testing. While software tests are designed to assert correct behavior of the system under a representative workload, fault injection asserts correct behavior under an additional *faultload*.

[3]https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html

3. Software fault injection—state-of-the-art

Fault injection is a versatile tool for dependability assessment. When an injected fault causes a system failure, this can indicate insufficient fault tolerance mechanisms. In general, the system can only be assumed robust and dependable when all its layers and components are fault tolerant and their fault tolerance has been verified empirically or formally.

Various fault injection implementation strategies with different characteristics exist. One classification thereof was given in Ref. [4] and is presented here in an adapted form.

Any fault injection tool relies on a **trigger mechanism** that causes the artificially generated fault or error to be inserted into normal program execution:

- *Time-based*: fault injection takes place at predetermined time intervals.
- *Location-based*: faulty values are written into predefined memory locations.
- *Execution-driven*: fault injection occurs dynamically, depending on the control flow.

While time-based fault injection can often easily be implemented non-intrusively, this is not the case for location-based fault injection, which is suitable for memory corruption, but impedes controlling the fault load dynamically. Execution-based fault injection allows for complex and realistic fault models, but is also not applicable to black box applications.

SFI can take place at different **injection times**:

- *Before runtime*: the program is modified upfront, for instance, by using source code mutation to add faults (software bugs) to the code.
- *During runtime*: faults are injected during program execution.
- *At the loading time of external components*: here, injection triggers may be the dynamic binding of external libraries or the adding of other dependencies during runtime.

Further, we distinguish between *synchronous* triggers, such as exception handling mechanisms, and *asynchronous* triggers, such as hardware interrupts.

Another question is which representations of a program, called **injection artifacts**, are to modify for fault injection purposes:

- Machine language or binary files
- Intermediate representations of the source code
- Source code

In general, as with other testing and profiling approaches, the behavior of a software system inevitably changes under faultload. To minimize this change, approaches should be as little intrusive as possible. The questions which fault injection approach to use, when to, and where to apply it using which faultload determine the effectiveness and efficiency of SFI.

3.1. Software fault injection approaches

A comprehensive survey of SFI techniques has been presented by Natella et al. [5]. This section presents selected SFI tools at different layers of abstraction within the software stack.

3.1.1. Operating system-level fault injection

The tool *Ballista* [6] has been used to compare different OS′ dependability by exercising system calls with both valid and invalid parameters. It bridges the gap between software testing and SFI by using test input selection methods, based on coverage criteria, to choose adequate fault injection targets.

Crashme[4] is a simple tool that tests the robustness of operating systems by attempting to execute random byte sequences as procedures. Despite its simplistic nature, crashme uncovered severe flaws in different operating systems and hypervisors.

Trinity[5] was a widely used "fuzzer"—i.e., a tool generating randomized inputs—for Linux system call interfaces, incorporating knowledge about file pointers and other OS-specific resource descriptors. The tool has been used to find many bugs in the Linux kernel.

3.1.2. Fault injection at interfaces

Targeting interfaces between software libraries, the SFI tool suite *execution driven fault injection (EDFI)* [4], enables the definition of fine-grained fault loads and extensible dynamic fault triggers. It relies on a combination of dynamic and static source code instrumentation.

Library fault injector (LFI) [7] injects faults at the interface between an application and its dynamically linked libraries. It applies binary analysis to obtain a fault model, describing which values and error codes can be returned from a C-style library and to analyze whether they are handled.

Further language-specific libraries for testing the robustness against interface faults exist, many of which are included in language runtimes, such as *Libfiu*[6] and Microsoft *TestApi*.[7]

3.1.3. Fault injection in distributed systems

In distributed systems, fault injection has a long history, mainly considering hardware and message passing in their failure cause models.

Orchestra [8] defines an architecture for inserting a protocol fault injection layer in the network stack, which applies filters to messages sent and received, to inject faults into them.

Chaosmonkey,[8] which has evolved into the *SimianArmy* resiliency testing tool suite, targets applications built upon AWS by making single-node instances unavailable. The approach of

[4]http://people.delphiforums.com/gjc/crashme.html
[5]https://codemonkey.org.uk/projects/trinity/
[6]https://blitiri.com.ar/p/libfiu/
[7]https://testapi.codeplex.com/
[8]https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

injecting faults into deployed systems was first successfully employed by Netflix and has been adapted by other companies.

With the advent of cloud computing, there has been a paradigm shift from trying to avoid failures at all costs to embracing faults as opportunities for making the system more resilient. The amount of effort put into the fault tolerance and resilience of cloud applications is often determined by service level agreements (SLAs), and the trade-offs between development effort, costs for redundancy, availability, and consistency are usually application-specific and based on management decisions.

3.2. Software failure cause models

Software failures are complex in their nature, origins, and manifestations. For SFI, a detailed understanding of the causality chains leading to software failure is needed. Software failure causes have been studied in various domains of software engineering research, providing a broad and heterogeneous spectrum of published models.

In a systematic literature study [2], we found that there is a lack of dynamic fault activation and error models. Such models are especially relevant for SFI. As shown in **Figure 1**, most of the papers—123 out of 156 publications—discuss a static, code-based fault model.

As software is becoming more and more crucial to dependability, so are software-focused failure cause models. Recent efforts to standardize knowledge about software failure causes include orthogonal defect classification (ODC) [9] and the common weakness enumeration (CWE) database.[9]

Additional layers of abstraction, as introduced, for example, by cloud management stacks, call for novel and more targeted failure cause models.

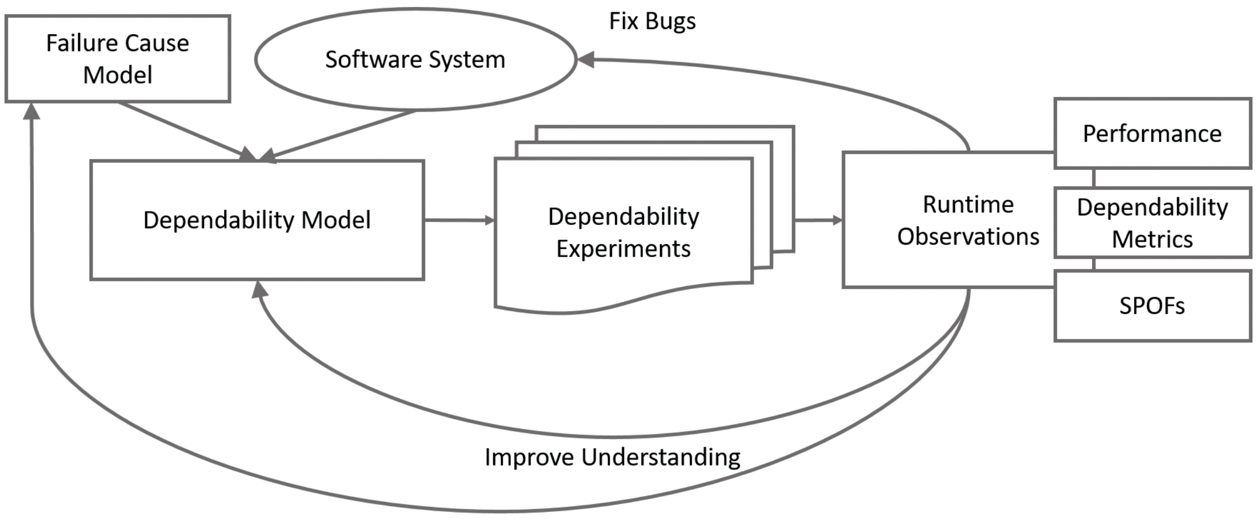

Figure 1. Literature study of failure cause models—most research targets static aspects.

[9]https://cwe.mitre.org

4. Problem statement

SFI is a versatile tool for dependability assessment and for testing the fault tolerance of increasingly complex systems. However, it is not yet widely used in real-world software systems outside the safety-critical and embedded domains. There is a gap between comparatively formal approaches, such as model-driven engineering, and what we choose to call *ad hoc software engineering*.

Formal approaches suffer from several limitations. They are successful in rather constrained domains, where full formal specification and fault modeling are possible. They are generally not applicable in modern scenarios involving rapidly changing requirements, under specified execution environments and fast-evolving software dependencies.

On the other hand, ad hoc software engineering is limited to implementing a software system according to some partial and informal specification. The resulting product is tested by its developers—usually without making strong coverage guarantees—and released without little other dependability assessment.

We assume that ad hoc software engineering constitutes most state-of-the-art software systems and is currently the most pragmatic approach in many situations, where fully formalized approaches are out of scope for technical as well as organizational reasons.

The following sections suggest how to improve ad hoc software engineering explicitly considering dependability aspects and by making software fault injection a pivotal dependability assessment means throughout the development process.

5. Methodology

We introduce *Fault Injection Driven Development* (*FIDD*), depicted in **Figure 2**, which integrates SFI into the development process. The following sections elaborate on its details.

5.1. Creating a dependability and failure cause model

To make the faultload introduced by SFI representative, two aspects need to be understood: The *dependability model* should capture the intended behavior in the presence of faults. This includes redundancy configurations, as well as error detection and recovery mechanisms. The *failure cause model* contains faults, errors, and fault activation conditions [10] that are anticipated and somehow addressed in the system architecture.

Modeling can serve to understand a complex system's dependability aspects to qualitatively discuss dependability bottlenecks or to quantitatively evaluate dependability metrics. Numerous modeling languages with varying expressiveness have been proposed. Here, we distinguish between *structural* and *behavioral* modeling languages.

Structural languages reflect the system structure typically at a higher level of abstraction such as components. The interactions and error propagation between components are in the

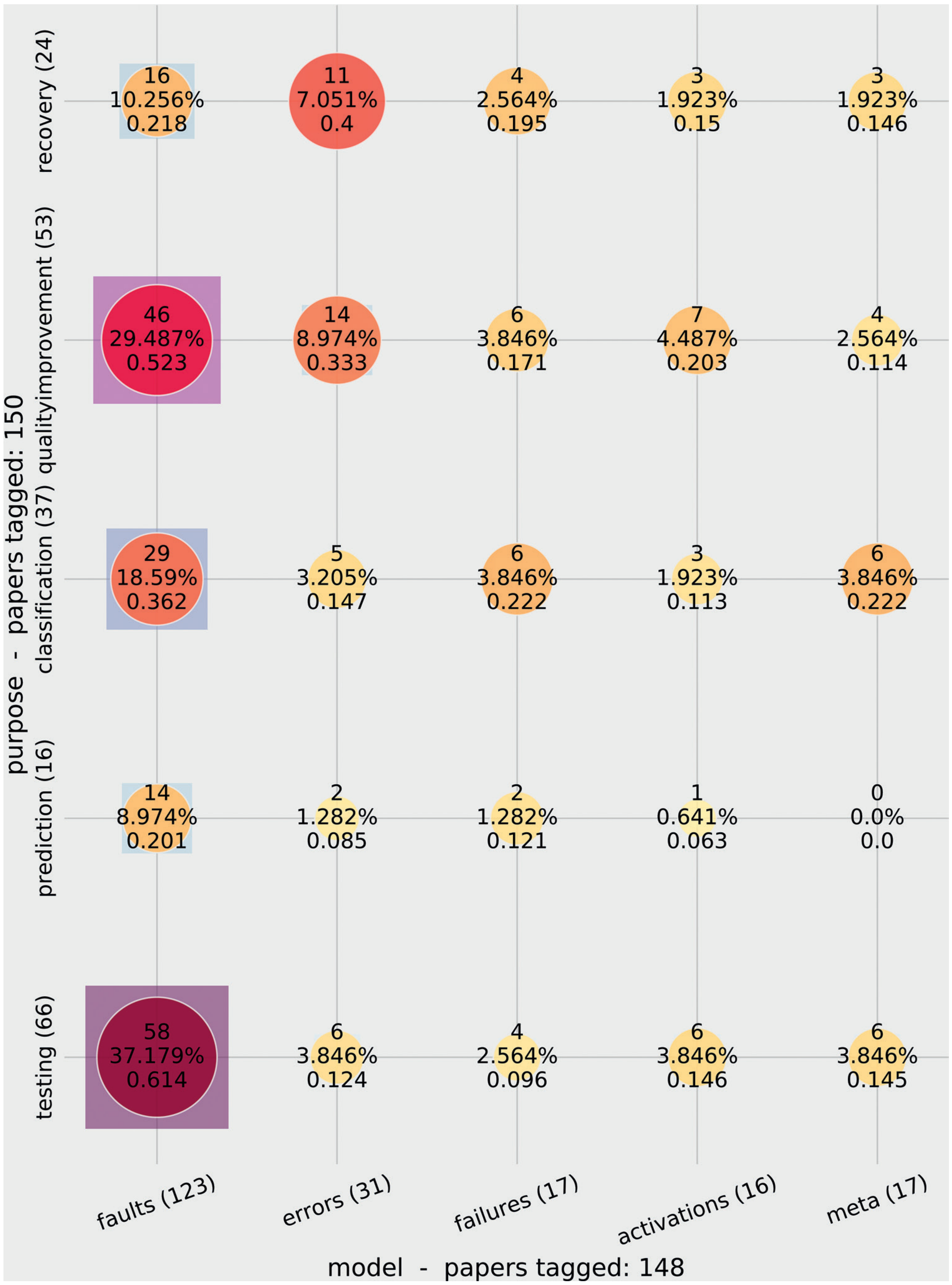

Figure 2. Overview of our fault injection–driven development methodology.

focus of such languages. Structural languages support human understanding of the system's dependability, but due to their coarse-grained nature, the mathematical expressiveness of such models is limited.

Behavioral dependability models describe the internal states of the system and their evolvement over time. They are fine-grained and suited for quantitative analyses. The choice of modeling language depends on the considered granularity, among other factors. Our focus lies on structural languages, for example, fault trees [11].

5.2. Generation of SFI campaigns

SFI experiments should be driven by user expectations and a structured understanding of the system assumptions, encapsulated in the dependability model. Here, we outline an approach, which creates an efficient and representative campaign of SFI experiments.

First, we use the dependability model to answer the question *where* to reasonably inject faults. Internal state changes or external events, which can contribute to fault activation, or directly cause a detectable error state, are called *fault injection points* (*FIPs*). They can be extracted from dependability models, for example, by listing all basic events of a fault tree.

Since software follows complex interaction patterns and fault tolerance mechanisms are largely sequence-dependent, the constellations of injected failure causes are critical. We believe the "one fault at the time" assumption, which is common with hardware fault injection, to be unrealistic for software systems, where synergistic effects are ubiquitous.

During a period of system runtime, a set of FIPs can be injected. Such SFI experiments add a certain *faultload* to the system. From the dependability model, we extract only experiments, which are expected to succeed. Experiments, which are not assumed to be tolerable, are less relevant, since they cannot verify whether the system adheres to its specification.

Considering the trade-off between test effort and coverage, we then find a reasonable balance by selecting a subset of experiments, while still attempting to cover all significant fault-tolerance mechanisms. Such a subset of experiments constitutes a *campaign*.

One selection criterion for experiments may be *maximality* in terms of induced faultload. The algorithm for applying this criterion is discussed in more detail in Ref. [12]. Maximal experiments may expose unforeseen synergistic effects. The significance of such effects has lately been acknowledged by safety-critical industries and their guiding standards [13].

5.3. Campaign execution

To make SFI practically applicable and thus more prominent in software development processes, its entire process requires automation. To achieve this, SFI automation frameworks are needed. These should be flexible, configurable, and suited for a broad range of applications. Based on user-provided or automatically generated fault injection campaigns, the experiments should be carried out automatically. Suitable performance and dependability metrics for analysis should be gathered.

In a previous work, we have presented two such SFI frameworks at different layers of abstraction:

Hovac [14] is an SFI framework for multithreaded applications, implemented in C++, which enables the customizable injection of various faults by using application programming interface (API) hooking, a nonintrusive technique requiring no source code access. The implemented faults and error states are based on the community-based CWE database.

We are currently also working on a tool suite, Faultmill, which fully automates the generation and orchestration of SFI campaigns in distributed systems. It relies on user-provided dependability models and provides a user-friendly interface for integrating custom fault injection scripts and executables.

5.4. Analysis and feedback

A remaining open question is which conclusions to draw from the results of campaigns, and how they can be leveraged to improve the overall process. If all experiments in the campaign succeed, this validates the initial dependability model. Furthermore, by gathering quality metrics, such as runtime or accuracy, the degradation under faultload can be quantified.

Known from software testing, *coverage measures* provide a quality metric for the assessment process itself. In a nutshell, they express how well a set of experiments (unit tests, SFI campaigns, or others) covers the vast space of possible behaviors. To provide feedback on the campaign, we suggest applying structural coverage criteria on the model.

6. Case study: SFI contest

The remainder of this chapter discusses a concise case study of applying SFI in software development teams: an SFI contest was carried out in a student seminar. Within the contest, we explored approaches and challenges of FIDD within a software development team. The contest took place during one semester within a course for 6 ECTS.

Students were divided into two teams, working on an application that renders a Bitmap image of the Julia fractal[10] in parallel using OpenMP.[11] Both teams were first asked to develop a fault model for this program, defining which types, frequencies, and severities of faults they were planning to implement in their fault injector. Subsequently, they were asked to specify the intended behavior of their own implementation under faultload.

The fault injection was implemented in a virtual machine (VM) running a (modified) Ubuntu Linux. VMs were configured using Vagrant[12] for portability. A fuzzing script was developed by us, which executed the students' fault-tolerant program within the fault-injecting VM environment, providing corner case and randomly generated input arguments.

[10]http://mathworld.wolfram.com/JuliaSet.html
[11]http://www.openmp.org/
[12]https://www.vagrantup.com/docs/getting-started/

6.1. Failure cause and dependability model

Each team first developed a *failure cause model* to be assessed with SFI. This process required some coordination between the teams to ensure that failure causes were on similar layers of abstraction. The resulting classes of failure cause are customized to a multithreaded C/C++ application running in a potentially unreliable execution environment:

- **Incorrect arguments**: unexpected parameters are passed to function calls or interfaces.
- **Memory errors**: the allocation of memory may fail or the memory may be corrupted.
- **Resource scarcity**: resources such as CPU time and file handles may be limited as well.
- **I/O errors**: I/O operations may be unreliable—they may fail, stall, or return errors.
- **Data type errors**: assumptions made on the data type or layout may be violated.
- **Erroneous system time**: the current time, queried by a system call, may be incorrect. This simulates time lags in distributed systems, as well as software or hardware flaws.

The intended system behavior under faultload was specified by both teams and took the failure cause model into consideration. While (due to time constraints) the specifications were not completely formalized but rather formulated in natural language, they offer some notable insights.

First, the specification contained *quantitative aspects,* such as "if dynamic memory allocation fails more than five times, we assume a permanent error, which is not tolerable." Moreover, many assumptions on the environment were made. This stresses the importance of environment-aware failure cause models. Finally, little focus was placed by students on performance and other quality metrics. This may indicate the necessity of more research and education regarding trade-offs among nonfunctional software properties.

6.2. Implementation of SFI

The developed SFI approaches were based on the mentioned assumptions:

System calls were intercepted using the *LD_PRELOAD*[13] environment variable. This variable was used to enforce the loading of SFI libraries, which offer identical interfaces to some function calls, such as *malloc*. Failing I/O operations were simulated analogously. To tolerate such injections, some critical functions, including the *new*-operator for C++, were re-implemented by the students.

The scarcity of resources, for example caused by other processes on the system, was injected by using the *cgroups*[14] facility. The application was limited in its CPU shares and memory. A second *cgroup* containing resource-intensive processes was created to put the application under stress. One group addressed this issue by snapshotting state to hard disk periodically.

[13]http://man7.org/linux/man-pages/man8/ld.so.8.html
[14]https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

In some scientific and distributed computations, an incorrect system time can lead to errors. Although this was not the case for the example application, the system time was manipulated using *libfaketime*.[15]

Bit-level memory errors were implemented by halting the program using *ptrace* and manipulating random memory areas. This injection of low-level faults turned out to be hard to predict and tolerate—it was addressed with redundant, checkpointed computations.

6.3. Lessons learned

In the contest, the most fault-tolerant application started five redundant processes, each checkpointing its state in short intervals. Under faultload, this application became so slow that the test scripts which were running it timed out.

There is an inherent trade-off between performance and fault tolerance, which induces runtime overhead for error detection, containment, and removal. This trade-off needs further analysis and specification.

In practice, the continuous integration of the SFI environment (a VM, submitted by the students) with the program template required a surprising amount of manual work due to software dependencies and hidden environmental assumptions. As full automation of SFI campaigns remains an engineering challenge, standardization of SFI interfaces is desirable.

As already mentioned, developing a dependability model and a failure cause model required a substantial amount of communication among the teams. While several efforts have been made, such as ODC and the CWE database, there is yet no established failure cause model, which is commonly known and used among software developers. Such a model would have been a powerful communication tool in the contest.

7. Discussion and conclusion

Software dependability is harder to ensure in complex systems with many layers of abstraction, interacting components, concurrency, and increased distribution.

SFI is a promising approach for evaluating the dependability of software systems, which scales even for large and complex systems, as the success of ChaosMonkey exemplifies. It is especially suited for ad hoc software engineering, where a formal understanding of the system is missing. Yet, SFI is not yet established in general software development practice.

We have introduced *Fault Injection Driven Development* (*FIDD*), a structured approach for applying SFI. We discuss how even starting without any specification or documented fault model, it is possible to make the dependability aspects explicit and create a fully automated environment, in which repeated fault injection campaigns assure dependability and yield relevant insights.

[15]https://github.com/wolfcw/libfaketime

The case study of an SFI student contest underlines the necessity of FIDD-like methodologies and provides insights into organizational and engineering challenges.

Many opportunities for further research remain: as discussed, most current failure cause models are not suited for SFI. They do not take dynamic aspects of software failures into account. However, dynamic fault activation and error models provide valuable input for SFI and are, therefore, topic of our ongoing research.

Practical experience shows that SFI frameworks are tedious to implement if they are to support full automation of campaign execution and extensibility. To support the reuse of such frameworks, some form of standardized interfaces for SFI would be immensely useful.

Acknowledgements

The authors would like to thank all participants of the student contest at HPI, during the winter semester 2016/17.

Author details

Lena Feinbube*, Lukas Pirl and Andreas Polze

*Address all correspondence to: lena.feinbube@hpi.de

Hasso-Plattner-Institute, University Potsdam, Germany

References

[1] Babbage C. Passages from the Life of a Philosopher. Longman, Green, Longman, Roberts, & Green; London; 1864

[2] Feinbube L, Tröger P, Polze A. The Landscape of Software Failure Cause Models. arXiv preprint arXiv:1603.04335. 2016

[3] Avižienis A, Laprie J-C, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing. 2004;**1**(1):11-33

[4] Giuffrida C, Kuijsten A, Tanenbaum AS. EDFI: A dependable fault injection tool for dependability benchmarking experiments. In: 2013 IEEE 19th Pacific Rim International Symposium on Dependable Computing (PRDC); 2013

[5] Natella R, Cotroneo D, Madeira HS. Assessing dependability with software fault injection: A survey. ACM Computing Surveys (CSUR). 2016;**48**(3):44

[6] Koopman P, Sung J, Dingman C, Siewiorek D, Marz T. Comparing operating systems using robustness benchmarks. In: Proceedings of the Sixteenth Symposium on Reliable Distributed Systems; 1997

[7] Marinescu PD, Candea G. LFI: A practical and general library-level fault injector. In: 2009 IEEE/IFIP International Conference on Dependable Systems & Networks; 2009

[8] Dawson S, Jahanian F, Mitton T. Orchestra: A Fault Injection Environment for Distributed Systems. In: Technical Report; 1996

[9] Chillarege R, Bhandari IS, Chaar JK, Halliday MJ, Moebus DS, Ray BK, Wong M-Y. Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on Software Engineering; 1992

[10] Tröger P, Feinbube L, Werner M. What activates a bug? A refinement of the Laprie terminology model. 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE); 2015

[11] Vesely WE, Goldberg FF, Roberts NH, Haasl DF. Fault Tree Handbook. DTIC Document; 1981

[12] Feinbube L, Pirl L, Tröger P, Polze A. Software fault injection campaign generation for cloud infrastructures. In: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion; 2017

[13] International Organization for Standardization. Road vehicles—Functional safety. 2011; (ISO 26262:2011(E))

[14] Herscheid L, Richter D, Polze A. Hovac: A configurable fault injection framework for benchmarking the dependability of C/C++ applications. In: Proceedings of the 2015 International Conference on Software Quality, Reliability, and Security; 2015