UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

A. D. MCCXXIV

Comunità Europea
Fondo Sociale Europeo

ACHIEVING REPRESENTATIVE FAULTLOADS IN SOFTWARE FAULT INJECTION

ROBERTO NATELLA

Tesi di Dottorato di Ricerca

(XXIV Ciclo)

Novembre 2011

Il Tutore

Prof. Domenico Cotroneo

Il Co-Tutore

Prof. Henrique Madeira (University of Coimbra)

Il Coordinatore del Dottorato

Prof. Franco Garofalo

Dipartimento di Informatica e Sistemistica

via Claudio, 21- I-80125 Napoli - ☎ *[#39]* (0)81 768 3813 - *[#39]* (0)81 768 3816

ACHIEVING REPRESENTATIVE FAULTLOADS IN SOFTWARE FAULT INJECTION

By
Roberto Natella

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
AT
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II
VIA CLAUDIO 21, 80125, NAPLES, ITALY
NOVEMBER 2011

Abstract

Given the complexity of modern software systems and its pervasiveness in many aspects of our lives, software faults (i.e., bugs) are a dangerous threat. Unfortunately, it is impossible to assure that software is perfect despite of advances in software engineering. Therefore, mission- and safety-critical systems have to provide fault tolerance algorithms and mechanisms to mitigate this threat. Software Fault Injection emerged in the last decades as a means for testing and improving fault-tolerant systems. This approach deliberately introduces faults in a software in order to assess its behavior in the presence of software faults.

In order to be adopted by practitioners in the development of critical systems, and to assure an effective and trustworthy evaluation of fault tolerance, the realism of faults being injected (*fault representativeness*) need to be assured, i.e., the injected faults should reflect the residual faults that escape the development process and that can affect the system. This thesis addresses fault representativeness with respect to three aspects. First, it proposes an approach for selecting code locations in which to inject software faults in a complex software system. The approach identifies locations in which faults are more likely to hide from testing, in order to focus the injection on the most representative locations and to reduce the number and cost of experiments at the same time. Second, it proposes a method for improving the accuracy of faults injected in binary code, which is required when the source code is not available as in the case of third-party software. Finally, this thesis proposes a technique for emulating concurrency faults, which are a significant part of faults affecting complex software. These contributions are instrumental to advance Software Fault Injection and make it an effective and practical approach for developing fault-tolerant systems.

Keywords: Software Faults, Fault Representativeness, Software Reliability, Fault Tolerance

Acknowledgements

I would like to thank Domenico Cotroneo for his support and guidance that greatly contributed to my professional and personal growth, and to professors Stefano Russo and Henrique Madeira, which are role models for me. I am also grateful to all my friends/colleagues of the MobiLab research group, of the CINI laboratory, and of the University of Coimbra for the time we enjoyed together. Last but certainly not least, I would like to thank Maria for her help, patience and love, and my family. Grazie!

Napoli, Italy Roberto

30/11/2011

Folklore in Software Engineering

Murphy's General Law
If something can go wrong, it will go wrong.

Murphy's Constant
Damage to an object is proportional to its value.

Naeser's Law
One can make something bomb-proof, not jinx-proof.

Troutman Postulates
1. Any software bug will tend to maximize the damage.
2. The worst software bug will be discovered six months after the field test.

Green's Law
If a system is designed to be tolerant to a set of faults, there will always exist an idiot so skilled to cause a nontolerated fault.

Corollary
Dummies are always more skilled than measures taken to keep them from harm.

Johnson's First Law
If a system stops working, it will do it at the worst possible time.

Sodd's Second Law
Sooner or later, the worst possible combination of circumstances will happen.

Corollary
A system must always be designed to resist the worst possible combination of circumstances.

Source: G.C. Buttazzo, *Hard Real-Time Computing Systems*, Kluwer Academic Publishers, 2nd Ed., 2010

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context and motivations

Our society is increasingly dependent on *software*, which is nowadays an integral part of devices and systems we interact with in our everyday life. Software-intensive systems range from small systems for entertainment and domotics, to large systems and infrastructures that provide fundamental services such as telecommunication, health, transportation, financial, and power supply systems. Given the pervasiveness of software in real-world scenarios, it becomes essential to assure that the software is *dependable*, that is, the software must be able to deliver service that can justifiably be trusted, in terms of readiness and continuity of correct service [16]. This is especially important in the case of *safety-critical* systems, in which a failure may cause death or injury to people, harm to the environment, or economical loss. These systems, and the software they include, are therefore required to be highly dependable.

Unfortunately, we are still far from achieving perfect software, due to two striving and opposing forces, which are the *growth of software complexity* and the *reduction of time and resources available for software development* [130]. On the one hand, the number and the complexity of software requirements and functionalities tend to increase, given the ambitious expectations on software of users and designers and the raise of awareness in its potentialities. An example of this growth is provided by NASA flight software [66]: the size of flight software used in space missions to Mars in terms of Lines of Code (LoCs), which is a rough yet indicative measure of complexity, increased exponentially from 5 thousands of LoCs (*Viking* mission, 1975) to 555 thousands of LoCs (*Mars Exploration Rover* mission, 2004). On the other hand, time and budget allocated to software development are finite and are kept at a minimum due to market constraints. These constraints limit the resources and reduce the effectiveness of rigorous analysis and design practices, which are intended to *prevent* the introduction of defects in the software product, and of verification and validation (V&V) activities, which aim to *detect and remove* defects (Figure 1.1). Market constraints also lead to the adoption of *reused* and *third-party software components*, such as *Commercial Off-The-Shelf* (COTS) components, which reduce the efforts required to develop and test whole parts of the software: however, this kind of components increase the likelihood of residual defects in the final product, since they are used in a new environment that was not previously foreseen, and they are significantly more difficult to test and debug due to the lack of source code and/or expertise on their internals and specification [204].

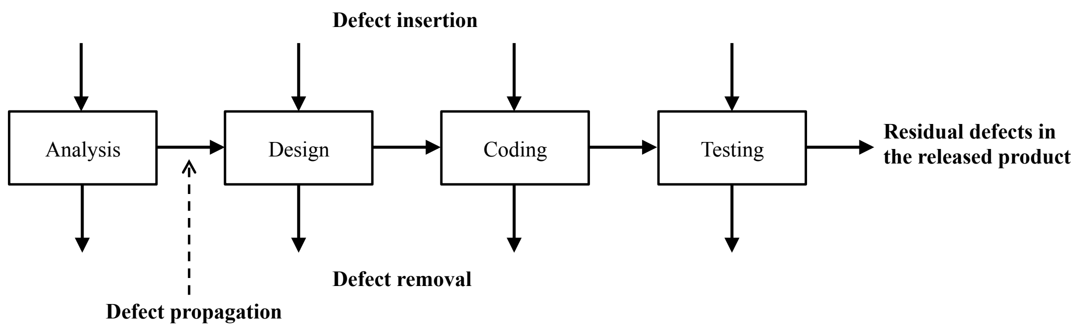


Figure 1.1: Insertion and removal of defects during the software development lifecycle [67].

As a result, the presence of *defects* in the software (which are also referred to as *software faults* or *bugs*) cannot be avoided in practice. Faulty software represents a threat to safety-critical systems and to their users, and it emerged as a major cause of failure in modern systems [83, 84, 122]. This fact is attested by the occurrence of several accidents caused by software faults, that provoked loss of a significant amount of money and even of human lives. In June 4th 1996, during the first test flight of the Ariane 5 rocket, the vehicle veered off its flight path and exploded less than one minute after take-off, causing a loss of half-billion dollars. The explosion was caused by a wrong data conversion in the software from 64-bit floating point to 16-bit signed integer representation. The bug resulted from the reuse of a software subsystem, without substantial re-testing, from the Ariane 4 mission, which developers assumed to be compatible with the new system [204]. Another software fault provoked, in August 14th 2003, the blackout of the General Electric energy management system, which left 50 million people in the northeastern America without power and cost around 6 billion dollars of financial loss. The bug affected an alarm and logging software system, and it was triggered by a unique combination of events and alarm conditions on the equipment being monitored. The failure of the alarm system led to a cascade of computer and equipment failures and to the blackout [79]. In 2002, a study commissioned by the US National Institute of Standards and Technology (NIST) estimated that the cost of software failures due to inadequate testing ranges between \$22.2 and \$59.5 billion [169].

The complexity of modern software and the limitations of software engineering force safety-critical systems to coexist with software faults, which will eventually provoke faulty conditions that have not been foreseen during testing. To face this problem, it is well known, even recommended by safety standards [4, 78], that software developers adopt *software fault tolerance* algorithms and mechanisms. Software fault tolerance is achieved by masking software faults through the adoption of diverse and redundant implementations of the same software (e.g., N-version programming, recovery blocks, N-self checking programming) [15, 132], and detecting a wrong state of the system, in order to provide a fail-stop behavior or a degraded mode of service (e.g., concurrent error detection, checkpointing and recovery, exception handling) [83, 84, 49].

In order to assess the effectiveness of fault tolerance mechanisms, and to gain adequate confidence that the system will be safe during operation, it is necessary to evaluate the system behavior under unforeseen faulty conditions. This can be done by deliberately injecting faults into the system. The process of introducing faults in a system in order to evaluate its behavior and to measure the efficiency (coverage, latency, etc.) of fault tolerance is referred to as *fault injection* [10, 37, 203]. It is recommended by software safety standards, such as the ISO/DIS 26262 standard for automotive safety [78], which prescribes the use of error detection and handling mechanisms in software and their verification through fault injection, and the NASA standard 8719.13B for software safety [4], which recommends fault injection to evaluate system behavior in the presence of faulty off-the-shelf software.

Many fault injection approaches have been proposed in the last decades. The first approaches consisted in injecting physical faults into the target system hardware (e.g., using radiation, pin-level, power supply disturbances, etc. [89, 10]). The growing complexity of the hardware turned the use of these physical approaches quite difficult or even impossible, and a new family of fault injection approaches based on the runtime emulation of hardware faults through software (*Software Implemented Fault Injection* - SWIFI) become quite popular, as it is generally well accepted that simple fault models such as bit-flip or bit stuck-at do represent real hardware faults [156, 114, 167, 12]. Some examples of SWIFI tools are Xception [28], NFTAPE [184], and GOOFI [6]. However, all these tools have been proposed for the emulation of hardware faults and their potential to emulate more complex faults such as software faults is very limited [133, 103].

The use of fault injection to emulate software faults, namely *Software Fault Injection*

(SFI), is relatively recent when compared to the first fault injection proposals [37, 203]. Unfortunately, in spite of decades of fault injection research, the fact is that the accurate emulation of residual software faults through fault injection remains largely unknown. Software faults are intrinsically difficult to be modeled, since they are caused by human mistakes occurring during the software development process (Figure 1.1). This reflects in the lack of understanding about how software faults affect a software artifact, and about how these faults can be introduced by intent in order to *emulate* the real software faults that are hidden in the software. The realism of the faults being injected, namely *fault representativeness*, is a key property of fault injection for assessing fault-tolerant systems. The faultload (i.e., the set of faults to be injected in a given software component/system) should reproduce the faults that are actually experienced in the field, that is, the *residual faults* that are overlooked by rigorous design and testing and that actually affect the mission of the system as evidenced by recent accidents in safety-critical systems, in order to obtain a realistic evaluation of fault tolerance in face of runtime faulty conditions. Fault representativeness is required to obtain accurate dependability measures, such as coverage factors of fault tolerance that take into account the likelihood of faults to exist in the field [163, 50, 161], and it is also important to analyze the complex failure modes that can be exhibited by a software system or component, which are not known a priori. If the injected software faults are not representative of residual faults, then it is risky to assert the effectiveness of software fault tolerance. The goal of this thesis is to evaluate and to improve the representativeness of faultloads in SFI, in order to achieve confidence in the assessment of fault-tolerant systems. This aim, in turn, is instrumental to make Software Fault Injection an effective and practical approach for developing fault-tolerant systems, and to favor its adoption in the software industry.

1.2 Open issues

Several attempts have been made towards the realistic injection of software faults. The first Software Fault Injection approaches adopted SWIFI to emulate the *effects* of software faults, by corrupting the system state (e.g., wrong or uninitialized pointers and flags, or an incorrect control flow). This is an indirect form of fault injection, as what is being injected is not the fault itself but only a possible effect of the fault. However, it is difficult to justify the representativeness of this type of injection, since it is difficult to relate errors with specific defects in the code.

Most recent approaches inject faults in a program by *changing its code.* They are based on a set of *fault types* that define which programming structures can be injected, and how they should be changed to introduce a realistic fault. The state-of-the-art is represented by the *Generic Software Fault Injection Technique* (G-SWFIT) [65], as its fault types are derived from the most frequent fault types found in widely-deployed complex systems in order to achieve fault representativeness.

One open issue, which has been neglected by existing approaches, is the **selection of**

fault locations in which to inject in terms of modules and/or routines. Complex systems have a huge number of locations in which to inject, and only few of them could be suitable to inject a realistic software fault. Existing techniques inject a fault type in every code location containing a given programming structure, without accounting for the complexity of code in the module/routine surrounding the code location and the testing efforts that have been spent on that part of code. This aspect is important for achieving fault representativeness, since it is well-known that residual software faults are not equally likely to exist in every code location, but their occurrence is **related to software complexity and to the testing activities performed during software development** [20, 201, 70]. Moreover, injecting defects in every location of complex software leads to a dramatic increase of the **cost of the SFI campaign**, in terms of number of experiments to be executed. For example, the software systems considered in this thesis have tenths of thousands of potential fault locations, and injecting faults in all of them could make impractical the fault injection campaign.

The injection of software faults by mutating the **binary executable code** of a program is another open issue in the field. SFI in binary code enables the **experimental dependability evaluation of systems for which the source code is not available**, which is often the case when third-party and COTS software is adopted. However, **assuring the accuracy of binary-level SFI is a major concern for its adoption in real-world**

scenarios. This kind of injection requires that programming constructs used in the source code are identified by looking only at the binary code, since the injection is performed at this level. Unfortunately, binary-level SFI is a difficult and error-prone task due to the complexity of programming languages and of modern compilers, which make difficult and in some cases impossible to accurately recognize where to inject faults. For this reason, it is important to assure that binary-level SFI is able to correctly emulate software faults to an acceptable degree.

Finally, an important but still unresolved issue is the injection of software faults related to timing and synchronization in concurrent programs (**concurrency faults**). These software faults manifest their effects in a non-deterministic way, depending on thread scheduling, timing of events, and interactions with the system state as a whole (hardware, OS, other software). For this reason, this kind of faults is referred to as *transient* software faults or "*Mandelbugs*". These faults are very difficult to detect and remove during software development, and they represent a significant share of residual faults in complex software systems due to the shift towards multithreaded software and multicore hardware architectures [83, 84, 34, 86, 35]. Nevertheless, these faults were overlooked by fault injection studies (e.g., this fault type was deliberately not encompassed in G-SWFIT) due to the relative scarcity of documentation about them, and to their complex nature and interactions with the environment (scheduling, timing, and interactions with the system state) that are **difficult to**

be accurately emulated through SFI.

1.3 Thesis contributions

This dissertation works on three directions to address the problem of fault representativeness in Software Fault Injection. More specifically, this thesis contributes to the state-of-the-art on Software Fault Injection by:

1. **Evaluating and improving the representativeness of injected fault locations**. This dissertation proposes an approach for selecting the most representative fault locations in a complex software, which takes into account the complexity and the relationships of software components, and the testing activities that are performed to remove faults, in order to better emulate residual faults that escape the software development process. An extensive experimental analysis (more than 3.8 millions of individual experiments in three real systems) shows that a significant share (up to 72%) of faults injected by G-SWFIT cannot be considered representative of residual software faults, as they are consistently detected by test cases, and that representativeness of injected faults is affected by the fault location within the system. The approach refines the faultload by selecting a subset of fault locations that are suitable for emulating residual software faults. This filtering is essential to assure meaningful results and to reduce the cost (in terms of number of faults) of software fault injection campaigns in

complex software. The proposed approach is based on classification algorithms, is fully automatic, and can be used for improving fault representativeness of existing Software Fault Injection approaches.

2. **Evaluating and improving the accuracy of faults injected at binary level**. In this dissertation, a method is proposed for assessing the accuracy of binary-level fault injection techniques and tools in large and complex systems, i.e., faults injected in the binary code correctly emulate software defects in the source code. This assessment is important to gain confidence that results from binary-level fault injection are accurate, and to test, debug and improve SFI techniques and tools with respect to real-world software. This thesis provides an extensive experimental evaluation of a R&D fault injection tool produced by Critical Software S.A.[1] based on G-SWFIT, in order to assess issues and limitations that are faced when injecting faults in a real-world complex software system (a satellite data handling system). More than 12 thousand binary-level faults in the OS and application code of the system were injected, and they were compared with faults injected in the source code by using the same fault types of G-SWFIT. The method was effective at highlighting the pitfalls that can occur in the implementation of G-SWFIT, and the limitations of this technique. The analysis

[1]This research has been made in the context of the European project *CRITICAL-STEP* ("CRITICAL Software Technology for an Evolutionary Partnership", `http://www.critical-step.eu`), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672 of the Seventh Framework Programme (FP7).

shows that G-SWFIT can achieve an improved degree of accuracy if these pitfalls are avoided.

3. **Extending Software Fault Injection to concurrency faults**. This dissertation proposes a technique for the injection of concurrency faults in multithreaded software. These faults, which are not encompassed by G-SWFIT, are a relevant part of residual faults in complex software. First, this thesis analyzes the limitations of G-SWFIT regarding its ability to emulate the transient nature of Mandelbugs, in the context of a fault-tolerant software system for Air Traffic Control[2]. It is found that injected faults do not realistically emulate Mandelbugs, since most of them are activated in the early phases of execution, and they deterministically affect process replicas in the system. Moreover, this behavior impacts on the verification of fault tolerance, as 35% of system states are not covered during the fault injection campaign. A technique is then proposed to emulate concurrency faults in a fully representative way, by controlling the environment conditions that make these faults to activate and manifest themselves. The analysis shows that controlling fault activation can also increase the confidence in fault tolerance, since it is possible to reduce the amount of untested states down to 5%.

[2]This research has been made in the framework of the private-public laboratory *COSMIC* ("Centro di ricerca sui sistemi Open Source per le applicazioni ed i Servizi MIssion Critical", DM23318, `http://www.cosmiclab.it`) and of the Italian research initiative *Iniziativa Software*, which involves the Finmeccanica group and several Italian universities (`http://www.iniziativasoftware.it`).

This thesis includes materials from the following research papers, already published in peer-reviewed conferences and journals or submitted for review:

- **R. Natella**, D. Cotroneo, H.S. Madeira, J.A. Duraes, *On Fault Representativeness of Software Fault Injection*, **IEEE Transactions on Software Engineering**, under minor revision
- **R. Natella**, D. Cotroneo, H.S. Madeira, J.A. Duraes, *Representativeness Analysis of Injected Software Faults in Complex Software*, **Proc. of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**, pp. 437-446, June 2010, Chicago, Illinois, USA, Acceptance Rate: 25% (PDS Track)
- D. Cotroneo, A. Lanzaro, **R. Natella**, R. Barbosa, *Experimental Analysis of Binary-Level Software Fault Injection in Complex Software*, **Proc. of the 9th European Dependable Computing Conference (EDCC)**, May 2012, Sibiu, Romania, under review
- **R. Natella**, D. Cotroneo, *Emulation of Transient Software Faults for Dependability Assessment: A Case Study*, **Proc. of the 8th European Dependable Computing Conference (EDCC)**, pp. 23-32, April 2010, Valencia, Spain, Acceptance Rate: 32%
- D. Cotroneo, **R. Natella**, *Fault Injection e Robustness Testing*, book chapter in *L'Analisi Quantitativa dei Sistemi Critici*, A. Bondavalli editor, pp. 233-270, 2011

The following research papers are related to this thesis but were not included. These papers adopt Software Fault Injection and Robustness Testing for the dependability evaluation and improvement of complex software systems:

- G. Carrozza, D. Cotroneo, **R. Natella**, A. Pecchia, S. Russo, *Memory Leak Analysis of Mission-Critical Middleware*, **Journal of Systems & Software (JSS)**, vol. 83, no. 9, pp. 1556-1567, 2010
- A. Bovenzi, G. Carrozza, M. Cinque, D. Cotroneo, **R. Natella**, *OS-Level Hang Detection in Complex Software Systems*, **Intl. Journal of Critical Computer-Based Systems (IJCCBS)**, Vol. 2, No. 3/4, pp. 352-377, 2011

- G. Carrozza, **R. Natella**, *A Recovery-Oriented Approach for Software Fault Diagnosis in Complex Critical Systems*, **Intl. Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)**, Vol. 2, No. 1, pp. 77-104, 2011
- D. Cotroneo, **R. Natella**, R. Pietrantuono, *Predicting Aging-Related Bugs using Software Complexity Metrics*, **Performance Evaluation–An International Journal**, under review
- M. Cinque, D. Cotroneo, **R. Natella**, A. Pecchia, *Assessing and Improving the Effectiveness of Logs for the Analysis of Software Faults*, **Proc. of Intl. Conference on Dependable Systems and Networks (DSN)**, pp. 457-466, June 2010, Chicago, Illinois, USA, Acceptance Rate: 25% (PDS Track)
- D. Cotroneo, D. Di Leo, **R. Natella**, R. Pietrantuono, *A Case Study on State-Based Robustness Testing of an Operating System for the Avionic Domain*, **Proc. of the 30th Intl. Conf. on Computer Safety, Reliability and Security (SAFECOMP)**, pp. 213-227, September 2011, Naples, Italy, 2011
- D. Cotroneo, **R. Natella**, R. Pietrantuono, S. Russo, *Software Aging Analysis of the Linux Operating System*, **Proc. of Intl. Symp. on Software Reliability Engineering (ISSRE)**, pp. 71-80, November 2010, San Jose, California, USA, Acceptance Rate: 32.3%
- D. Cotroneo, **R. Natella**, S. Russo, *Assessment and Improvement of Hang Detection in the Linux Operating System*, **Proc. of Intl. Symp. on Reliable Distributed Systems (SRDS)**, pp. 288-294, September 2009, Niagara Falls, New York, USA, Acceptance Rate: 28%, 2009
- D. Cotroneo, **R. Natella**, R. Pietrantuono, S. Russo, *Software Aging and Rejuvenation: Where we are and where we are going*, **Third Intl. Workshop of Software Aging and Rejuvenation (WoSAR)** (co-located with ISSRE 2011)
- D. Cotroneo, **R. Natella**, R. Pietrantuono, *Is Software Aging related to Software Metrics?*, **Second Intl. Workshop of Software Aging and Rejuvenation (WoSAR)** (co-located with ISSRE 2010)
- D. Cotroneo, D. Di Leo, **R. Natella**, *Adaptive Monitoring in Microkernel OSs*, **DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance (PFARM)** (co-located with DSN 2010)

- M. Cinque, **R. Natella**, A. Pecchia, S. Russo, *Improving FFDA of Web Servers through a Rule-Based Logging Approach*, **Intl. Workshop on Field Failure Data Analysis (F2DA)** (co-located with SRDS 2009)
- D. Cotroneo, **R. Natella**, A. Pecchia, S. Russo, *An Approach for Assessing Logs by Software Fault Injection*, **DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance (PFARM)** (co-located with DSN 2009)
- G. Carrozza, D. Cotroneo, **R. Natella**, A. Pecchia, S. Russo, *An Experiment in Memory Leak Analysis with a Mission-Critical Middleware for Air Traffic Control*, **ISSRE Intl. Workshop of Software Aging and Rejuvenation (WoSAR)** (co-located with ISSRE 2008)
- D. Cotroneo, M. Cinque, G. Carrozza, **R. Natella**, *Operating System Support to Detect Application Hangs*, **Second Intl. Workshop on Verification and Evaluation of Computer and Communication Systems (VeCOS)**, July 2008, Leeds, UK

The thesis is organized as follows. Chapter 2 provides basic concepts on fault injection, and surveys previous relevant work on the emulation of software faults and its applications to the assessment and improvement of dependable systems. Chapter 3 presents an empirical assessment of the problem of fault representativeness in real-world complex software, and the proposed approach for improving fault representativeness by careful selection of fault locations. Chapter 4 provides a method for assessing the accuracy of binary-level fault injection, along with lessons learned from the assessment of an industrial fault injection tool. Chapter 5 focuses on the emulation of concurrency faults, and presents the proposed approach and results obtained from an Air Traffic Control system. The dissertation concludes with final remarks and the indication of the lesson learned and future research directions.

Chapter 2

Software Fault Injection: Background and related work

2.1 Introduction

Many threats affect computer systems dependability, that originate from the several parts that compose a system, including hardware and software faults, and from the environment and from people and other systems that interact with the computer system during operation. The characterization of faults and of the way they can affect the dependability of critical computer-based systems is a complex subject, and it is instrumental for understanding how we can improve the dependability of these systems and our confidence that dependability can be justifiably trusted. This characterization is especially important in the context of fault injection, in order to reproduce faults into an experimental setting and to achieve a realistic assessment of fault-tolerant systems. Our focus (Figure 2.1) is on the characterization and the emulation of *software faults*, which are a complex and still not well understood class of faults, that emerged in the last decades as a major cause of failures. Therefore, significant efforts have been spent towards the characterization of software faults, with the aim to prevent them during software development activities, and to design and to assess fault-tolerant systems able to coexist with their unavoidable occurrence.

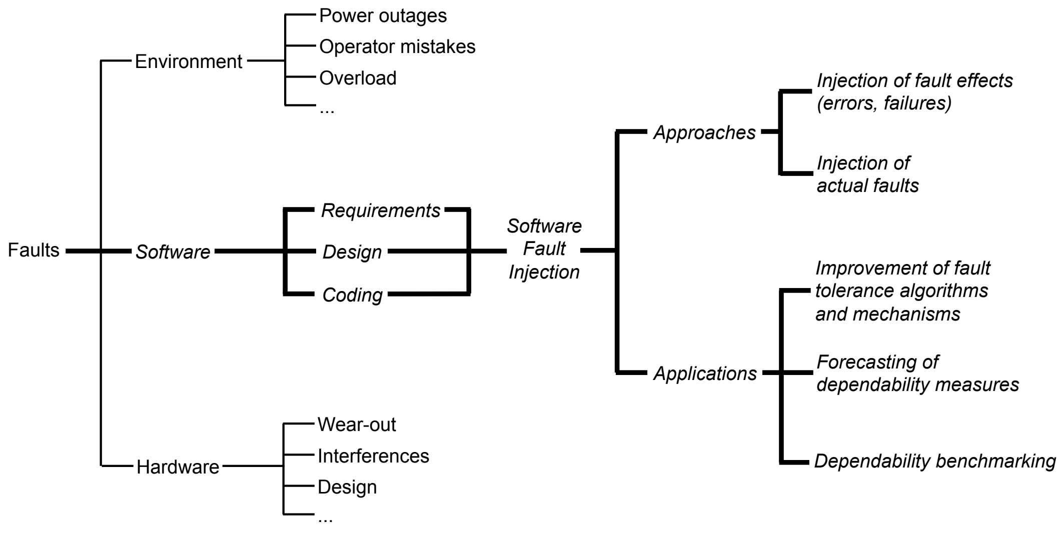

Faults
Environment
Power outages
Operator mistakes
Overload
...
Software
Requirements
Design
Coding
Hardware
Wear-out
Interferences
Design
...
Software Fault Injection
Approaches
Injection of fault effects (errors, failures)
Injection of actual faults
Applications
Improvement of fault tolerance algorithms and mechanisms
Forecasting of dependability measures
Dependability benchmarking

Figure 2.1: Scope of this thesis.

This chapter first provides some preliminary concepts about fault injection, which are the foundations of Software Fault Injection. Subsequently, the characterization of software faults is discussed, along with some key results that emerged from the empirical analysis of software faults in real software systems. In doing so, we give emphasis to the problem of artificially emulate these faults by Software Fault Injection. We then survey several techniques and tools that have been proposed in the past for emulating software faults, and how they evolved to achieve fault representativeness, non-intrusiveness, repeatability, practicability and portability in Software Fault Injection experiments. Finally, relevant applications of Software Fault Injection are described, with the aim to highlight which kinds of result can be obtained from Software Fault Injection, and how it can support the design of fault-tolerant systems.

2.2 Basic concepts and definitions

According to [16], a *fault* is the adjudged or hypothesized cause of an incorrect system state, which is referred to as *error*. A *failure* is an event that occurs when an incorrect service is delivered, that is, an error state is perceived by the users or external systems. These concepts have clearly a great relevance for fault injection, as the results of fault injection in a system reflect the class of faults that have been injected. There are many kinds of fault that can affect computer systems, which can be grouped in hardware, software, and environment faults (Figure 2.1). Fault injection can be adopted to assess the reaction of a system with respect to any of these fault classes. Therefore, fault injection is usually divided in *Hardware Fault Injection*, *Software Fault Injection*, and *Environment Fault Injection*. A *fault model* is a formal description that specifies the set of faults that the system is expected to experience during operation, and that are injected to assess and validate fault tolerance. The fault model depends on the system requirements, the environment in which the system will operate, and the development process that has been adopted.

Although there are several fault injection approaches and fault models, it is possible to recognize a common conceptual schema of fault injection [95], shown in Figure 2.2. The system under analysis is usually referred to as *target*. There are two entities that stimulate the system, respectively the *load generator* and the *injector*. The former exercises the target with inputs that will be processed during a fault injection experiment, while the latter introduces a fault in the system. The set of inputs and faults submitted to the system are respectively referred to as *workload* and *faultload*, which are typically specified by the tester in a descriptive form (*library*) to the load generator and to the injector. The introduction of a fault is performed by tampering with the structure or the state of the system or with the environment in which it executes. Fault injection usually involves the execution of several *experiments* or *runs*, which form a *fault injection campaign*, and only one or few faults from the faultload are injected during an individual experiment.

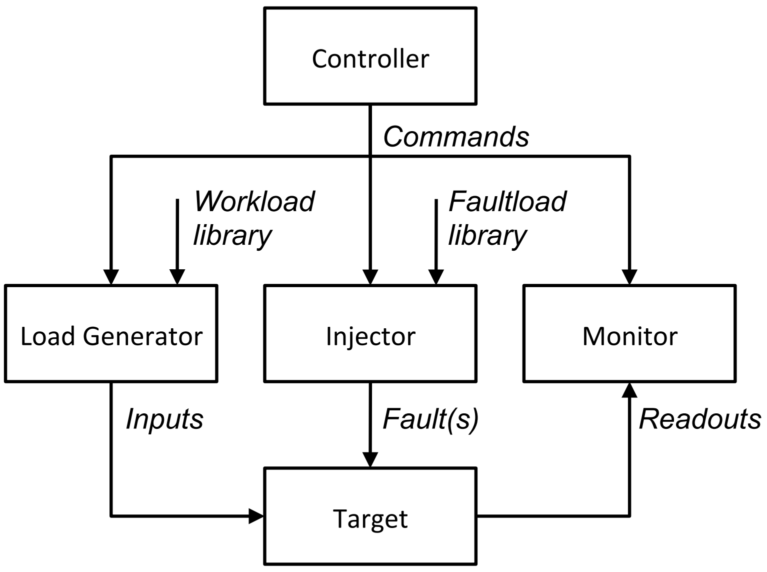


Figure 2.2: Conceptual schema of fault injection [95].

The *monitor* entity collects from the target the raw data (*readouts* or *measurements*) that are needed to evaluate the effects of injected faults. The choice of readouts depends on the kind of system considered and on the properties that have to be evaluated. They may include the outputs of the target (e.g., messages sent to users or to other systems) and the internal state of the target or its parts (e.g., the contents of a specific area of memory). Readouts are used to assess the outcome of the experiment: for instance, the tester can extrapolate whether the injected fault has been tolerated, or the system has failed. In order to obtain information about the outcome of an experiment, readouts are usually compared to the readouts obtained from fault-free experiments (also referred to as *golden runs* or *fault-free runs*). Finally, the *controller* entity coordinates the other entities. Typically, the controller is also responsible for iterating through several fault injection experiments that form the fault injection campaign, and to store the results of each experiment to be used for subsequent analysis.

Figure 2.3 depicts the states of a fault injection experiment. A fault is injected when the target is in a correct state. The target stays in a correct state until the fault causes an error: in this case, the fault has been *activated*, or *triggered*. For instance, fault activation may occur when an external input stimulates a part of the system affected by the fault. An error may *propagate*, that is, other portions of the target state get corrupted, until a failure occurs. It may also happen that the target returns in a correct state, since the error can be masked or corrected by redundant devices and logic (*fault tolerance*), or erroneous data can be overwritten by correct data or reinitialized. A fault injection experiment can terminate without producing a failure: either the fault has not been activated (the fault is *dormant*) or an error is present but it has not become a failure (the error is *latent*).

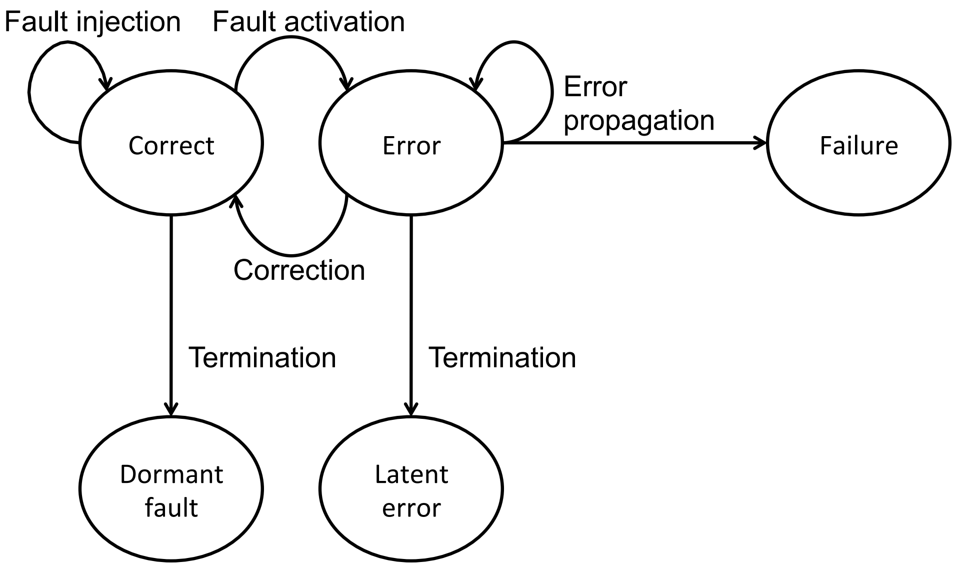


Figure 2.3: States of a fault injection experiment.

Another form of fault injection is represented by *error injection*, in which the *effects* of faults are introduced (in place of the hypothesized faults that may cause them) by corrupting the state of the target. This approach accelerates the occurrence of failures, since it avoids to wait for fault activation. Moreover, error injection can be cheaper or more feasible than fault injection: this is the case of *Software-Implemented Fault Injection* (SWIFI), in which the effect of hardware faults (e.g., CPU or memory faults) are reproduced by corrupting the state of the software, instead of physically tampering with hardware devices.

The results collected from a fault injection experiment can be used to obtain *measures* that characterize the behavior of the target system in the presence of faults. There exist several kinds of measures and only few of them are usually relevant, depending on the fault model and the kind of target system. In general terms, these measures describe the system from the point of view of its *ability to tolerate faults*. In fact, the sole presence of fault tolerance algorithms and mechanisms does not guarantee that a system cannot fail, and fault injection is a means to assess and improve fault tolerance. The effectiveness of a fault tolerance mechanism or algorithm, namely the *coverage* [26, 163, 50], is defined as

$$c = \Pr\{\, H = 1 \mid (f, a) \in F \times A \,\}\ , \tag{2.1}$$

that is, the conditional probability that a fault is correctly handled ($H = 1$) given the occurrence of a fault f and a sequence of inputs a. In other terms, the coverage factor c can be viewed as the expected value of a discrete random variable H that take the values 0 or 1 for each pair (f, a), denoted with $h(f, a)$. The coverage can be expressed in an equivalent way as

$$c = \sum_{(f,a) \in F \times A} h(f, a) \cdot p(f, a)\ , \tag{2.2}$$

where $p(f, a)$ denotes the relative probability of occurrence of (f, a). Another measure of fault tolerance effectiveness is the *mean coverage time* (also referred to as *latency*) [10, 11], that is, the expected value τ of a random variable T_p representing the time required to handle a fault, which can expressed as

$$\tau = E[T_p] = \int_0^\infty t \cdot f_{T_p}(t)dt \; . \tag{2.3}$$

where $f_{T_p}(t)$ denotes the probability density function of T_p.

Deficiencies in fault tolerance can be due to development faults affecting the design or implementation of fault tolerance algorithms and mechanisms (*error and fault handling coverage* [14, 26]), or to fault assumptions differing from the faults faced during operation (*fault assumption coverage* [162]), that is, incorrect assumptions about the behavior of a failed component or about the independence between component failures.

Fault injection is adopted for coping with these deficiencies, specifically for the *fault removal* and *fault forecasting* of fault tolerance [16, 10]. In the case of fault removal [17], fault injection is used to define test cases, in which faults represent the inputs of test cases: the results of this analysis are qualitative and are used to improve the implementation of fault tolerance algorithms and mechanisms. In the case of fault forecasting [10, 11], fault injection is aimed at quantitatively estimating the effectiveness of fault tolerance that will be provided during operation, to provide feedback to the development process on the design of fault tolerance and to compare the effectiveness of different designs. Fault injection can assess coverage factors and latency (i.e., the time needed for recovering from a fault) of fault tolerance, which in turn can be used in analytical models to obtain dependability measures (e.g., availability, performability). Other measures of interest for fault forecasting are related to the occurrence of specific failure modes, such as the probability of a *fail-stop* behavior and the probability of catastrophic failures from the point of view of *safety*. For both the fault removal and fault forecasting goals, it is necessary or desirable to achieve a set of key properties [10, 24]:

- **Representativeness** refers to the ability of the faultload and the workload to represent the real faults and inputs that the system will experience during operation. In the case of fault removal, this property is important to avoid focusing development efforts towards tolerating faults that cannot realistically occur in practice. This property is also important for fault forecasting, since it is a necessary condition to assure that the estimates obtained from experiments reflect the operational behavior of the system. Representativeness of faultloads is achieved by defining a realistic fault model, and by accurately reproducing this fault model when faults are injected.

- **Non-intrusiveness** requires that the instrumentation adopted in the fault injection process (such as fault insertion and data collection) should not significantly alter the behavior of the system, and that perturbations on the obtained measures are kept at a minimum. For instance, intrusiveness can be caused by the execution of additional code in the case of software-implemented fault injection.

- **Repeatability** is the property that guarantees statistically equivalent results when a fault injection campaign is executed more than once using the same procedure in the same environment. This property is not trivial to achieve due to the many sources of non-determinism in computer systems, such as thread scheduling and timing of events, and it requires to isolate the fault injection environment from external disturbances.
- **Practicability** refers to the effectiveness of fault injection in terms of cost and time. These factors include the time required to implement and setup the fault injection environment, the time to execute the experiments, and the time for the analysis of results. This property requires that experiments are supported by tools and can achieve an high degree of automation, in order to fulfill the time and budget constraints of software development.
- **Portability** requires that a fault injection technique or tool is applicable with low effort to different systems, in order to allow their comparison. The portability of a fault injection tool also refers to the ability of the tool to support several fault models or to be extended with new fault models.

2.3 Characterization of software faults

Software faults, on which this thesis focuses, are a inherently complex class of faults since they are human mistakes affecting the requirements, design, and implementation of the software. According to the taxonomy of Avizienis et al. [16], software faults can be defined as *permanent* faults in the software that occur during its *development*[1]. Due to the complex nature of software development, it is not trivial to characterize how software faults originate during the several phases of the development process (including requirement analysis, high-level and low-level design, coding, and even testing), and in which way they affect the structure of the software and its behavior during execution. Software faults are the result of the process and technologies adopted by developers, and they also depend on the kind of system being considered.

This complexity reflects in the many efforts that have been spent for understanding and characterizing software faults in order to better cope with them. This section reviews some models of software faults that have been proposed in past work, each with different purposes and in different contexts, and discusses how these models relate with fault injection.

[1]It should be noted that the term *error* is also adopted by the software engineering community and in the IEEE Standard Glossary of Software Engineering Terminology [1] to refer to the concept of *software defect*. This thesis complies to the taxonomy of Avizienis et al. [16], in which the term *error* refers to an incorrect system state produced by a software defect.

2.3.1 IEEE Standard Classification for Software Anomalies

The analysis of software faults occurring during the software lifecycle is widely adopted for assessing and improving the software development process. Data about software faults are used for identifying and mitigating the root causes of software faults, such as by adopting better practices and improving the scheduling and assignment of project activities. The analysis of the software development process is recommended by software quality standards and frameworks such as the Capability Maturity Model Integrated (CMMI) [40], and it is particularly important for the development of safety- and mission-critical software, which rely on rigorous processes in order to assure that an acceptable safety level is achieved. However, several kind of data can be collected for this purpose, which can vary across organizations.

The IEEE Standard Classification for Software Anomalies (published in 1994 [2] and revised in 2010 [5]) is a document providing guidance on how to characterize software faults. It provides a set of mandatory *defect* and *failure* attributes that are nowadays commonly collected and used (in this form or a similar one) by software development processes and supporting tools, such as Bugzilla[2] and JIRA[3], for the purposes of problem tracking and process improvement. In its original version [2], the standard enumerated in detail the

[2]`http://www.bugzilla.org/`
[3]`http://www.atlassian.com/software/jira/`

possible *defect types*, which included logic problems (e.g., forgotten cases or steps, iterating loop incorrectly, misinterpretation), computation problems (e.g., equation insufficient or incorrect, sign convention), interface/timing problems (interrupts handled incorrectly, subroutine/module mismatch), and others. However, the enumeration of software fault types is affected by several limitations. Although the standard is built on the experience of several experts, it is impossible to assure that the list of fault types is complete, and that there exists no further fault type. Moreover, many of the fault types are related to a specific technology or type of system (e.g., faults in interrupt handling are related to OSs; incorrect equation faults are related to control systems), and are not applicable in many cases.

The standard does not include anymore a classification of fault types in its latest revision [5], and provides a set of attributes (Table 2.1) without mandating a set of fault types. In a similar way, it provides attributes for software failures (Table 2.2) for supporting the analysis of field failure data and the resolution of problems at the customer site. However, this kind of characterization still leaves open the problem of characterizing software faults and providing a related fault model for dependability assessment.

2.3.2 Orthogonal Defect Classification

The Orthogonal Defect Classification (ODC) [36] is a framework for classifying software faults in order to obtain measurements and feedback from the development process. ODC represents a trade-off between classical quantitative approaches used for reliability growth

Table 2.1: Defect attributes in the IEEE Standard Classification for Software Anomalies.

Attribute	**Definition**
Defect ID	Unique identifier for the defect.
Description	Description of what is missing, wrong, or unnecessary.
Status	Current state within defect report life cycle.
Asset	The software asset (product, component, module, etc.) containing the defect.
Artifact	The specific software work product containing the defect.
Version detected	Identification of the software version in which the defect was detected.
Version corrected	Identification of the software version in which the defect was corrected.
Priority	Ranking for processing assigned by the organization responsible for the evaluation, resolution, and closure of the defect relative to other reported defects.
Severity	The highest failure impact that the defect could (or did) cause, as determined by (from the perspective of) the organization responsible for software engineering.
Probability	Probability of recurring failure caused by this defect.
Effect	The class of requirement that is impacted by a failure caused by a defect.
Type	A categorization based on the class of code within which the defect is found or the work product within which the defect is found.
Mode	A categorization based on whether the defect is due to incorrect implementation or representation, the addition of something that is not needed, or an omission.
Insertion activity	The activity during which the defect was injected/inserted (i.e., during which the artifact containing the defect originated).
Detection activity	The activity during which the defect was detected (i.e., inspection or testing).
Failure reference(s)	Identifier of the failure(s) caused by the defect.
Change reference	Identifier of the corrective change request initiated to correct the defect.
Disposition	Final disposition of defect report upon closure.

Table 2.2: Failure attributes in the IEEE Standard Classification for Software Anomalies.

Attribute	**Definition**
Failure ID	Unique identifier for the failure.
Status	Current state within failure report life cycle.
Title	Brief description of the failure for summary reporting purposes.
Description	Full description of the anomalous behavior and the conditions under which it occurred, including the sequence of events and/or user actions that preceded the failure.
Environment	Identification of the operating environment in which the failure was observed.
Configuration	Configuration details including relevant product and version identifiers.
Severity	As determined by (from the perspective of) the organization responsible for software engineering.
Analysis	Final results of causal analysis on conclusion of failure investigation.
Disposition	Final disposition of the failure report.
Observed by	Person who observed the failure (and from whom additional detail can be obtained).
Opened by	Person who opened (submitted) the failure report.
Assigned to	Person or organization assigned to investigate the cause of the failure.
Closed by	Person who closed the failure report.
Date observed	Date/time the failure was observed.
Date opened	Date/time the failure report is opened (submitted).
Date closed	Date/time the failure report is closed and the final disposition is assigned.
Test reference	Identification of the specific test being conducted (if any) when the failure occurred.
Incident reference	Identification of the associated incident if the failure report was precipitated by a service desk or help desk call/contact.
Defect reference	Identification of the defect asserted to be the cause of the failure.
Failure reference	Identification of a related failure report.

models and qualitative ones for defect root cause analysis. Reliability growth models provide a statistical characterization of a software product and its development process, by identifying trends in the number of detected faults or field failures [146], although these trends are identified in the late phases of the software lifecycle and can provide little feedback to the development process. Qualitative approaches consist in the investigation of causes of individual defects by a specialized team in order to define actions that can prevent defects, although this investigation may require significant resources and it is not intended to provide statistical trends or for quantitative analysis.

ODC provides *quantitative means* for gaining insights on the development process, by obtaining inferences from the analysis of the empirical distributions of defects. Differing from reliability growth models, defects are divided among different classes and the per-class distribution along the development process is analyzed. The division among classes is motivated by the observation that there is a cause-effect relationship between the development process and the kind of defects found during development [38]. ODC proposes a set of *defect types* based on the *defect fix* made by the programmer that corrects the fault (Table 2.3). The main benefit of this classification is that defect types can be associated with the activities of the different stages of development. For instance, the occurrence of a significant number of Function defects can point out that the development process should be improved in the high-level design phase. Defect types are associated to different process stages (high-

Table 2.3: ODC defect types.

Defect type	**Definition**
Assignment	Value(s) assigned incorrectly or not assigned at all.
Checking	Missing or incorrect validation of parameters or data in conditional statements.
Algorithm	Efficiency or correctness problems that affect the task and can be fixed by (re)implementing an algorithm or local data structure without the need for requesting a design change.
Timing/Serialization	Necessary serialization of shared resources is missing, wrong resource has been serialized or wrong serialization technique employed.
Interface	Communication problems between users, modules, components or device drivers and software.
Function	Affects a sizeable amount of code and refers to capability that is either implemented incorrectly or not implemented at all.

and low-level design, coding, documentation, high- and low-level design inspections, code inspections, unit/function/system test), and their distribution measured along stages is exploited to provide feedback on the process. Another benefit of ODC is that the classes are mutually exclusive (i.e., orthogonal) and that they are close to the programmer, since they are based on the correction of the defect: therefore, defects can be easily and unambiguously classified, and measurements lend themselves to quantitative analysis. Several pilot projects confirmed the usefulness of ODC to provide feedback [36].

ODC also proposes a classification scheme for providing insights on the verification process [36, 31]. To this aim, it defines a set of *defect triggers*, that is, the condition(s) allowing a defect to surface during V&V activities or in the field (Table 2.4, Table 2.5 and Table 2.6). Defect triggers are independent from defect types: for instance, Checking defects can be triggered when the system is executed under a new hardware platform, as well as when the

Table 2.4: ODC defect triggers for system tests.

Defect trigger	**Definition**
Startup/Restart	The system was being initialized or restarted due to an earlier shutdown or failure.
Workload volume/Stress	The system was operating near some resource limit, either upper or lower.
Recovery/Exception	An exception handler or a recovery procedure was active. The fault would not have surfaced if some earlier exception had not invoked the handler or the procedure.
Hardware/Software configuration	Triggers related to unusual or unanticipated configurations.
Normal mode	A trigger that says that no special conditions must exist for the fault to surface, i.e., the system was working well within upper and lower resource limits.

system activates a recovery procedure. Ideally, the distribution of triggers of field defects should be similar to the distribution of defects found by system tests (Table 2.4). If this is not the case, the difference can highlight issues in system tests. In a similar way, the distribution for document review and code inspection (Table 2.6) can point out weaknesses in the review process: this feedback is useful to assign reviews to people with skills in a specific product area.

The concept of defect type revealed to be quite useful for formulating a model of software faults, since defect types are unambiguous and directly related to the code. The approach proposed in [41] defines a faultload representative of software faults occurring in the field, based on failure data of the system under analysis. The faultload reflects the proportion of defect types found in the field, and errors are injected associated to these defect types. Subsequent field data studies [61, 65] extended the analysis of defect types to several software systems (including user/OS and young/mature software written using the C language). An

Table 2.5: ODC fault triggers for unit and function testing.

Defect trigger	**Test model**	**Definition**
Simple path coverage	White-box	The test case that found the defect was created by the tester with the specific intention of exercising branches in the code.
Combinatorial path coverage	White-box	The tester attempted to invoke the execution of branches under several different conditions.
Side effects	White-box	The defect surfaced because of some unanticipated behavior not specifically tested for.
Test coverage	Black-box	The test case that found the defect was a straightforward attempt to exercise a single body of code using a single input.
Test sequencing	Black-box	The test case that found the defect executed, in sequence, two or more bodies of code each of which can be invoked independently by the tester.
Test interaction	Black-box	The test case that found the defect initiated an interaction between two or more bodies of code each of which can be invoked independently by the tester. The interaction was more involved than a simple sequence of the executions.
Test variation	Black-box	The test case that found the defect was a straightforward attempt to exercise a single body of code using different inputs.
Side effects	Black-box	The defect surfaced because of some unanticipated behavior not specifically tested for.

Table 2.6: ODC fault triggers for document review and code inspection.

Defect trigger	**Definition**
Design conformance	The document reviewer or the code inspector detects the defect while comparing the design element or code segment being inspected with its specification in the preceding stage(s).
Understanding details	The inspector detects the defect while trying to understand the details of the structure and/or operation of a component.
Backward compatibility	The inspector used extensive product experience to determine an incompatibility between the functionality described by the design document or the code and that of earlier versions of the same product.
Lateral compatibility	The inspector with broad-based experience detected an incompatibility between the functionality described by the design document or the code and the other (sub)systems and services with which it must interface.
Rare situation	The inspector used extensive experience or product knowledge to foresee some system behavior not considered or addressed by the documented design or code under review.
Document consistency/completeness	The defect surfaces because of some inconsistency or incompleteness within the document or code.
Language dependancies	The developer detects the defect while checking the language-specific details of the implementation of a component or a function.

important finding from these analyses is that defect types follow the same trend across field data studies and software systems (Table 2.7): Algorithm defects are the largest part, Assignment and Checking defects have approximately the same weight, and Interface and Function defects are the less frequent ones. This result has an important implication on fault injection: the distribution of ODC defect types is independent from the particular system, therefore field failure data about the system under analysis (which are usually not available) are not needed to define a faultload for fault injection.

Table 2.7: Comparison between distributions of ODC defect types of two field data studies.

ODC defect type	**Distribution**	
	[65]	[41]
Assignment	21.1%	21.98%
Checking	25.0%	17.48%
Interface	7.3%	8.17%
Algorithm	40.1%	43.41%
Function	6.1%	8.74%

Furthermore, the field data study in [65] looked in detail at the faults in order to achieve a more precise characterization. To this aim, ODC defect types were extended to provide additional details and relate the faults with the *programming language construct* that is either *missing*, *wrong*, or *extraneous* (Table 2.8). In this context, a programming language construct is any building block of the program, such as statements and expressions. This classification is more oriented towards fault injection, since it gives an indication on how to manipulate a program in order to introduce a fault (for instance, the construct to be removed in order to inject a *missing* construct fault).

Table 2.8: Refined fault types from ODC defect types [65].

ODC type	Nature	Examples	# faults	% of total
Assignment	Missing	A variable was not assigned a value, a variable was not initialized, etc.	62	9.3%
	Wrong	A wrong value (or expression result, etc.) was assigned to a variable	70	10.5%
	Extraneous	A variable should not have been subject of an assignment	11	1.6%
Checking	Missing	An *if* construct is missing, part of a logical condition is missing, etc.	113	16.9%
	Wrong	Wrong logical expression used in a condition in branch and loop construct	53	7.9%
	Extraneous	An *if* construct is superfluous and should not be present	1	0.1%
Interface	Missing	A parameter in a function call was missing; incomplete expression was used as parameter	11	1.6%
	Wrong	Wrong information was passed to a function call (value, expression result, etc.)	38	5.7%
	Extraneous	Surplus data is passed to a function	0	0%
Algorithm	Missing	Some part of the algorithm is missing (e.g., function call, a iteration construct, etc.)	222	33.2%
	Wrong	Algorithm is wrongly coded or ill-formed	40	6.0%
	Extraneous	The algorithm has surplus steps; a function was being called	6	0.9%
Function	Missing	New program modules were required	21	3.1%
	Wrong	The code structure has to be redefined to correct functionality	20	3.0%
	Extraneous	Portions of code were completely superfluous	0	0%

The refined fault types have then been used to classify 668 faults from the field [65]. The analysis found that few fault types account for most of the faults, and that the remaining fault types each encompasses a small number of faults. Therefore, the study identified a set of fault types to be considered as being representative of faults in the field, based on two criteria: (i) the number of occurrences of the fault type must be at least as high as the average, and (ii) the occurrences should not be restricted to only one or two of the programs. The fault types fulfilling these criteria are reported in Table 2.9. This set of fault types represent a total of 67.6% of all faults collected. The study argues that these types should be addressed in the first place when emulating software faults. It also notes that although other fault types may occur in the field, they are probably very rare since they were not observed among the considered field faults, and would not affect the analysis of the most frequent fault types.

This analysis represents an important result towards the characterization of software faults: it identified the fault types that are likely affect a software system during operation, and provides a *generic* empirical distribution of fault types (i.e., it holds for several software systems). These fault types are applicable to other procedural programming languages, since they are not tied to specific features of the C language. Other studies were made to extend this fault model to the Java object-oriented language [21, 173]: they found that the fault types in Table 2.9 are still the most frequent ones, and that they can be extended with a set of object-oriented fault types when an object-oriented language is considered.

2.3.3 Bohrbugs, Mandelbugs, and Aging-Related Bugs

Another dimension which is often adopted for characterizing software faults is represented by the *fault activation reproducibility*, since this dimension has important implications on software fault tolerance. Reproducibility refers to the ability to identify and replicate the

Table 2.9: Most common fault types found in several software systems [65].

	Fault types	#	**ODC defect type**				
			Ass.	**Chk.**	**Int.**	**Alg.**	**Fun.**
Missing	*if* construct plus statements	71				✓	
	AND sub-expr in expression used as branch condition	47		✓			
	function call	46				✓	
	if construct around statements	34		✓			
	OR sub-expr in expression used as branch condition	32		✓			
	small and localized part of the algorithm	23				✓	
	variable assignment using an expression	21	✓				
	functionality	21					✓
	variable assignment using a value	20	✓				
	if construct plus statements plus *else* before statements	18				✓	
	variable initialization	15	✓				
Wrong	logical expression used as branch condition	22		✓			
	algorithm - large modifications	20					✓
	value assigned to variable	16	✓				
	arithmetic expression in parameter of function call	14			✓		
	data types or conversion used	12	✓				
	variable used in parameter of function call	11			✓		
Extraneous	variable assignment using another variable	9	✓				
Total		**452**	**93**	**135**	**25**	**192**	**41**
Coverage relative to each ODC type (%)		**68%**	**65%**	**81%**	**51%**	**72%**	**100%**

activation pattern of a fault that had caused one or more errors. Faults that are easily reproducible are called *solid*, or *hard*, faults, otherwise they are called *elusive*, or *soft*, faults [16]. Fault activation reproducibility explains the fact that although software faults are permanent in nature, many software failures exhibit a transient behavior and are difficult to diagnose, since it is difficult to reproduce and analyze the chain of events that exposed the fault during testing or production.

This fact has been experienced by programmers since a long time [87], and appeared in print for the first time in a seminal work of Jim Gray [83]. In that work, Gray hypothesized that most production software faults are soft, since industrial software goes through rigorous design and testing (structured design, design reviews, quality assurance, alpha test, beta test) and years of production that get rid of hard faults. In analogy to the Heisenberg Uncertainty Principle, soft faults are referred to as *Heisenbugs*, since these faults do not manifest themselves when trying to debug them due to perturbations introduced by debuggers (e.g., initialization of unused memory, influence on CPU scheduling and timing of events). Conversely, solid bugs are referred to as *Bohrbugs* (in analogy to the Bohr atom model), since they are easy to diagnose once detected. In the most recent taxonomies of software faults, Heisenbugs are included in the more general class of *Mandelbugs*, where the former are the bugs that change their behavior when probed using a debugger, and the latter encompass all the bugs whose activation condition is related to timing and to complex interactions with the system state as a whole, including hardware, operating system, and other software such as middleware, virtual machines, libraries and remote services [86].

The distinction between Bohrbugs and Mandelbugs is relevant for architecting fault-tolerant software systems (Figure 2.4). Bohrbugs always produce a failure every time the failed operation is repeated, and can only be tolerated by adopting *design diversity*, that is, different design/implementations of the same functionality are used to mask faults in individual implementations [129]. Conversely, Mandelbugs can be tolerated by reinitializing the software state and retrying the failed operation, because their activation conditions tend to disappear due to their transient nature [83, 34, 88]. This finding enables to devise fault-tolerant architectures that are more cost-effective, since several software failures can be avoided without the additional costs of the design diversity, in which the same functionality is implemented several times by different teams and using different processes and technologies. It is also advisable to detect and remove Bohrbugs during development via systematic and thorough testing [88].

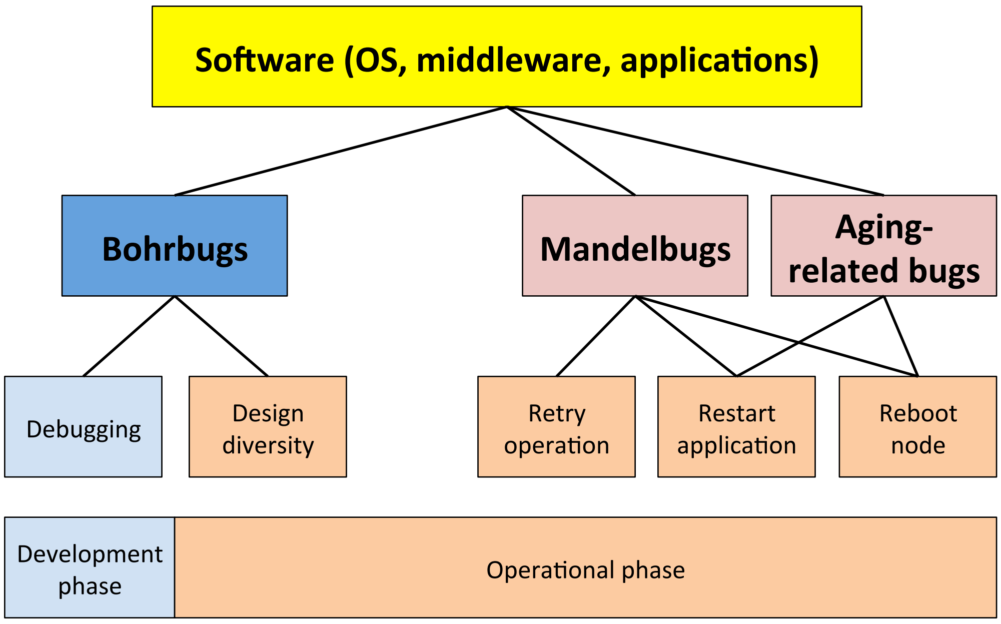


Figure 2.4: Relationships between software fault tolerance and Bohrbugs, Mandelbugs, and Aging-Related Bugs [192].

Several field data studies found evidence that Mandelbugs account for a significant part of complex software systems, although the share of Mandelbugs varies with the kind and maturity of system. The analysis of field failures in Tandem computer systems [84, 126] shown that most of software failures were caused by Mandelbugs and were successfully tolerated by *process pairs*, in which a critical function is replicated on two processing units, respectively a primary and a backup process that replaces the primary when its failure is detected. More recent studies found that although Bohrbugs represent the majority of software faults, Mandelbugs account for a significant share (in the 20-40% range) [34, 86, 35]. Figure 2.5 shows the proportion of Bohrbugs in NASA missions [86]: this proportion seems to stabilize around almost the same value for all the considered missions. This result emphasizes the importance of Mandelbugs in mission-critical systems. At the same time, it highlights that even in mission-critical and well-tested software a large proportion of Bohrbugs can still be present in the operational system, and that testing strategies have still room for improvement in order to reduce the number of Bohrbugs in complex software.

It is worth mentioning that an important subclass of faults of Mandelbugs has been identified by field data studies, namely *Aging-Related Bugs*. These bugs are the root cause of the *software aging* phenomenon, in which software systems running continuously for a long time tend to show degraded performance and an increased failure occurrence rate [96]. This kind of fault causes the accumulation of internal error states, or their activation and/or error propagation is influenced by the total time the system has been running [86]. In early

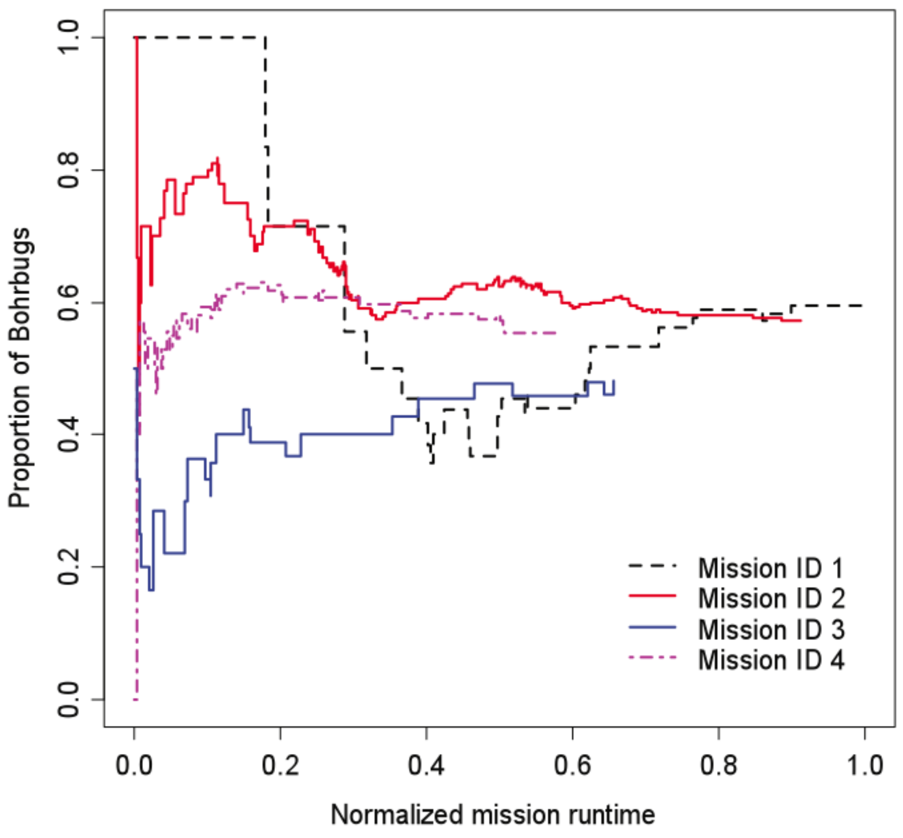


Figure 2.5: Proportion of Bohrbugs in NASA missions [86].

taxonomies, Aging-Related Bugs were overlapped both to Bohrbugs and to Mandelbugs, depending on the influence of the environment on the faults [195]. More recent studies framed these faults within Mandelbugs [87, 88], since they manifest themselves only after a long execution time and therefore their activation and/or error propagation should be considered complex. For their nature, these faults can be handled by proactively bringing the system to a state free from aging errors before an aging failure occurs (Figure 2.4): this approach is referred to as *software rejuvenation*. An example is represented by system reboot, which brings the system to its initial state. Software rejuvenation can improve system availability since the the downtime due to scheduled maintenance is lower than downtime due to unexpected failures. It should be noted that software aging revealed to be an issue for many long-running software systems (such as web servers [85], telecommunication systems [18], military systems [88], middleware [180, 30], and operating systems [47]), and that Aging-Related Bugs represent a non-negligible share of faults even in mission-critical software (4.4% of faults found in NASA missions were actually aging-related [86]).

Even if the distinction between Bohrbugs and Mandelbugs has a great relevance for fault-tolerant systems, it has not been carefully taken into account in past work on fault injection. Most error injection approaches implicitly or explicitly assumed that Mandelbugs generate transient errors analogous to SWIFI error models [19, 160]. However, no evidence is available assuring that the injected errors emulate Mandelbugs. Conversely, fault injection approaches based on changes in the program code do not encompass fault types related to Mandelbugs, such as the Timing/Serialization ODC type, and do not try to reproduce the complex activation and error propagation dynamics that characterize Mandelbugs. This aspect is still an open issue for fault injection, and part of this thesis focuses on this issue.

2.4 Software Fault Injection techniques and tools

The emulation of software faults has been pursued in several ways, and many techniques and tools have been developed in more than 20 years. We here illustrate and discuss these efforts, by distinguishing between two fundamental approaches: the injection of *faults effects* (also referred to as error injection), in which an error is introduced by perturbing the system state, and the injection of *actual faults*, in which the program code is changed in order to emulate a software fault in the code. The following subsections review Software Fault Injection techniques and tools, respectively: (i) the earliest approaches for ”data error injection” (subsection 2.4.1) that were based on hardware fault injection techniques existing at that time; (ii) approaches for ”interface error injection” (subsection 2.4.2), that were specifically aimed at testing the robustness of components with respect to interactions with other components; (iii) approches for the injection of actual faults, that introduce small faulty changes in the program code (subsection 2.4.3).

2.4.1 Data error injection

The early approaches for the injection of fault effects have grown in the context of studies on hardware faults through Software-Implemented Fault Injection (SWIFI). SWIFI aims at reproducing the effects (i.e., errors) of hardware faults (such as CPU, bus, and memory faults) by perturbing the state of memory or hardware registers through software. SWIFI is a low cost and easy-to-control approach for injecting hardware faults, that overcomes several problems associated with physical fault injection techniques (e.g., pin-level injection [10] and heavy-ion radiation [89]), such as controllability and repeatability of experiments [12]. This approach has been adopted for Software Fault Injection with the assumption that injected errors are representative of errors generated by software faults.

SWIFI approaches replace the contents of a memory location or register with a corrupted value. In order to perform this operation, three aspects need to be defined (see also Table 2.10):

- **What to inject.** This aspect is related to the kind of errors to be injected, by replacing a correct value with an incorrect one. SWIFI tools modify the contents of an individual bit, byte, or word in a memory location or register. Several error types have been defined from the analysis of errors generated by faults at the electrical or gate level [114, 167, 156]. For example, one common error type is the replacement of a bit or byte with a fixed value (*stuck-at-0* and *stuck-at-1* faults) or with the opposite value (*inverted faults*).
- **Where to inject.** There are many locations in memory or register banks that can be targeted by SWIFI. Errors injected in memory typically target random locations, due to the large amount of potential memory location. The selection of memory locations can be focused on specific memory areas (e.g., stack, heap, global data) or user-selected locations (e.g., a specific variable in memory). Errors injected in registers can target those registers that are accessible through software (e.g., data and address registers).
- **When to inject.** The instant in which an error is injected can be *time* or *event*

Table 2.10: SWIFI error models used for CPU, bus and memory faults.

What to inject	*Bit/byte set*, *Bit/byte reset*, *Bit/byte toggle*, *AND/OR/XOR with a user-defined bitmap*
Where to inject	*Stack memory*, *Heap memory*, *Global data memory*, *User code*, *Kernel code*, *User-defined memory location*, *Data registers*, *Address registers*, *Stack pointers*, *Program counters*, *Status register*
When to inject	*First access to memory location or register*, *Every access to memory location or register*, *Random time*, *User-defined time*

dependent. In the former case, an error is injected after that a given experiment time is elapsed, where the time is selected by the user or through a random distribution. In the latter case, the error is injected when a specific event occurs during execution, such as at the first access or at every access to the target location. These approaches are adopted for emulating three types of hardware faults, respectively *transient* (i.e., occasional), *intermittent* (i.e., recurring several times), and *permanent* faults.

It should be noted that hardware errors injected by SWIFI tools can be injected both in the program state (e.g., data and address registers, stack and heap memory) and in the program code (e.g., in memory areas where code is stored, before or during program execution). This is a relevant distinction for Software Fault Injection. Corruptions in the program state aim to reflect the *effects* of software faults, i.e., an error caused by the execution of a faulty program, such as a wrong pointer, flag, or control flow. SWIFI tools can introduce this kind of errors in a straightforward way. Corruptions in the program code aim to reflect *actual* software faults in the code, although the adoption of SWIFI tools for this purpose is not trivial, and it is discussed later in the context of ”code changes” approaches (subsection 2.4.3).

Early studies on SWIFI focused on the design of techniques and tools with the aim to provide a flexible and non-intrusive support to fault injection. The same error model was adopted for both hardware and software faults (Table 2.10). The Fault Injection based Automated Testing environment (*FIAT*) [177, 19] was one of the first SWIFI tools aimed at the injection of errors caused by both hardware and software faults. Errors are injected in a OS process through the support of *library routines* that are linked to the task executable during compilation, and alter process memory on request from an external program. Library code is also adopted for observing the process behavior and collect data. This solution is aimed to overcome OS protection mechanisms that prevent external processes from modifying the state of a target process. A limitation of this approach is represented by the need for the object code or the source code of the target, since object/source code may not be available in the case of third-party and COTS software, and additional effort is required to link the library and recompile the target. The FIAT system also provides an user interface and an hardware/software architecture (Figure 2.6) for assisting the definition of workload and faultload libraries, for controlling remote experiments and for data analysis (e.g., computing the coverage of error detection).

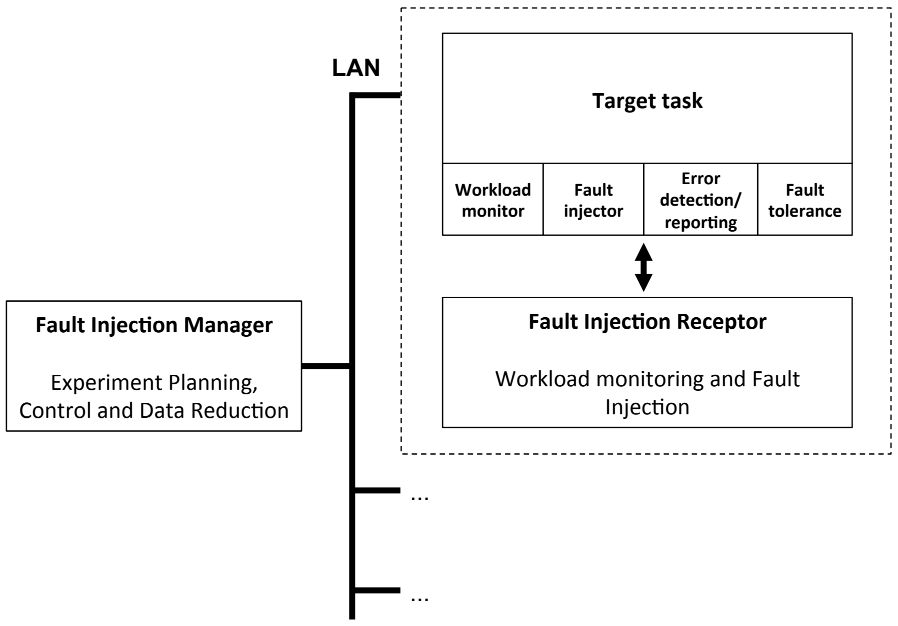

LAN
Target task
Workload monitor
Fault injector
Error detection/ reporting
Fault tolerance
Fault Injection Manager
Experiment Planning, Control and Data Reduction
Fault Injection Receptor
Workload monitoring and Fault Injection
...
...

Figure 2.6: The Fault Injection based Automated Testing environment (*FIAT*) [177].

In order to overcome the need for the object/source code, the Fault and ERRor Automatic Real-time Injector (*FERRARI*) [113, 115] SWIFI tool injects errors by corrupting the binary executable of the target program before execution, and by corrupting the in-memory image of a process during execution. The latter mode is achieved through the *ptrace* POSIX system call, provided by UNIX operating systems for debugging purposes, that allows a process to read and to write the memory of another process. This system call is used by an "error injection process" to control the execution of a target process, in a similar way to a debugging tool (Figure 2.7): it interrupts the execution of the target when a given instruction is executed (by replacing the instruction with a "software trap" instruction) or a given time elapsed, injects an error while the target is stopped, and lets the target execute again in order to observe its behavior in the presence of an error. Compared to FIAT, this approach does not require the object/source code of the target, although it is more intrusive since the "error injection process" and the software trap mechanism introduce some context switches (which can still be considered a negligible overhead for most applications).

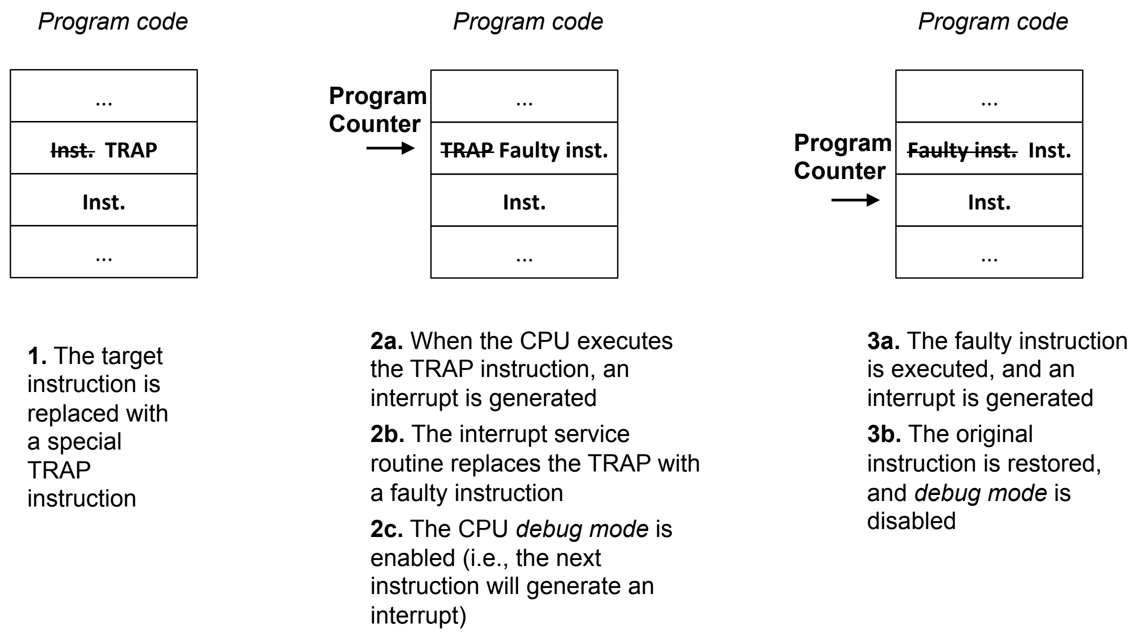

Program code
Program code
Program code
...
~~Inst.~~ TRAP
Inst.
...
Program Counter
...
~~TRAP~~ Faulty inst.
Inst.
...
Program Counter
...
~~Faulty inst.~~ Inst.
Inst.
...
1. The target instruction is replaced with a special TRAP instruction
2a. When the CPU executes the TRAP instruction, an interrupt is generated
2b. The interrupt service routine replaces the TRAP with a faulty instruction
2c. The CPU *debug mode* is enabled (i.e., the next instruction will generate an interrupt)
3a. The faulty instruction is executed, and an interrupt is generated
3b. The original instruction is restored, and *debug mode* is disabled

Figure 2.7: Software trap mechanism adopted by the Fault and ERRor Automatic Real-time Injector (*FERRARI*) [115].

The integrateD sOftware fault injeCTiOn enviRonment (*DOCTOR*) [91] SWIFI tool extended the SWIFI approach for emulating communication faults (see Table 2.11 summarizes this error model). The injection of communication faults takes advantage of the *x-kernel*, which is a OS kernel in which DOCTOR is implemented. This OS allows to introduce a layer between any two protocol layers in the protocol stack, in order to perform additional

Table 2.11: SWIFI error models used for communication faults.

What to inject	*Lose message(s)*, *Duplicate message(s)*, *Alter message header*, *Alter message body*, *Delay message(s)*
Where to inject	*Faulty link*, *Faulty direction*, *Single message*
When to inject	*Random time*, *User-defined time*, *User-defined message*

processing on any network message. A fault injection layer is inserted below the protocol or user program to be tested, and it intercepts operations between the target and protocol at lower levels. For instance, in order to cause a delayed outgoing message, the fault injection layer stops a message and schedules a future message with the same contents to be sent later. This approach was also adopted by the *ORCHESTRA* fault injection environment [52, 51], which introduced a programming support (*script-driven probing*) for specifying message filters able to inject faults in complex protocols (e.g., a fault is injected can be injected in a specific protocol state). Memory and CPU faults are injected using the software trap mechanism, with the exception of permanent CPU faults that are injected by replacing assembly instructions in the executable.

Since it was observed that many experiments produce latent errors that do not affect the target system, the Fault Tolerance And Performance Evaluator (*FTAPE*) [194, 193] was developed for improving the efficiency of SWIFI in terms of experiments that actually exercise fault tolerance mechanisms. It combines SWIFI with a workload generator and a workload activity monitoring tool (*MEASURE*), and injects errors in a system component

(CPU, memory, or disk) based on the amount of stress that the workload places on each component, exploiting the fact that errors are more likely to propagate and produce a failure behavior in the presence of a stressful workload (*stress-based injection*). FTAPE generates synthetic CPU, memory and disk operations, and it monitors the actual workload activity to determine the time and location for fault injection (Figure 2.8). CPU and memory faults are emulated in a similar way to other SWIFI tools; disk faults are emulated using a modified device drivers that returns SCSI and disk errors. A *path-based injection* approach was also proposed [193], that is based on the preliminary analysis of resource usage (e.g., CPU registers and memory locations) in order to injects faults in time periods during which a resource is ”live” and being used. The FTAPE tool has been engineered from scratch in subsequent efforts, with the aim to provide greater flexibility (i.e., the ability to introduce new fault models in the tool) and portability (i.e., most of the tool code can be reused when a new platform is targeted), and to support distributed systems. These efforts led to the Networked Fault Tolerance And Performance Evaluator (*NFTAPE*) [184], which introduced the concept of *LightWeight Fault Injector*, i.e., small programs to be invoked by NFTAPE in order to inject a fault, that embeds the logic needed to implement a fault model for a given target platform. Using this approach, NFTAPE was able to successfully perform fault injection on several platforms using several fault models.

SWIFI techniques and tools evolved accordingly with the increase in complexity and

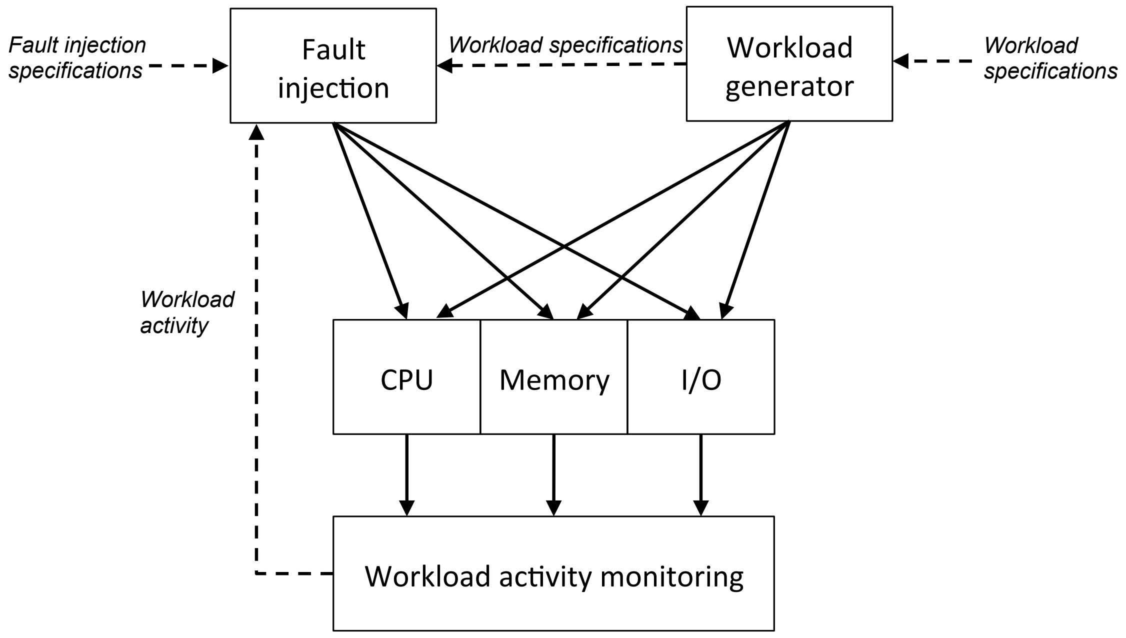


Figure 2.8: The Fault Tolerance And Performance Evaluator (*FTAPE*) [194].

functionalities of the underlying hardware. The *Xception* tool [29, 28] was developed to exploit the debugging and performance monitoring features existing in most modern CPUs, in order to allow a less-intrusive and more flexible fault injection and execution monitoring without increasing the cost of the experimental setup. Xception makes use of these features to improve the software trap approach adopted by previous tools. A *breakpoint register* is setup in order to trigger an *exception handler* when a given instruction is executed by the target. Using this feature, context switches are avoided since the handler is implemented and executed as a kernel routine, with a significant reduction of intrusiveness. This approach can inject errors in both user and kernel memory. Moreover, breakpoint registers also allow to inject faults when a specific data address is loaded or stored, which was not possible using the software trap mechanism (the only way to trigger a fault was the access to a specific instruction address where the software trap is placed), therefore extending the flexibility of SWIFI. Finally, *performance counters* in the CPU are used for time-based fault triggering, by registering an exception handler to be executed after a given number of processor clock cycles have elapsed, and for obtaining time measures such as error latency. Performance counters are more accurate than the system clock and can reduce time uncertainties of the measures. The Generic Object-Oriented Fault Injection (*GOOFI*) [6] tool is another SWIFI tool based on hardware support commonly available in modern systems. In order to inject errors at run-time, it makes use of *boundary* and *internal scan chains*, that is, built-in test-logic present in modern VLSI circuits to send and read data and instructions to integrated circuits and their interconnections in a hardware board. This approach is referred to as Scan-Chain Implemented Fault Injection (*SCIFI*). GOOFI provides an object-oriented software architecture for achieving flexibility and portability, and has been implemented using a platform-independent language (Java) and an SQL DBMS for storing and analyzing experimental data.

The main issue behind the approaches previously described is that they overlook the representativeness of errors with respect to software faults, by assuming that software faults cause state perturbations similar to hardware faults (e.g., bit-flips). This assumption is questionable since the state manipulations performed by a faulty program can be much different than those caused by random physical faults, such as wear-out and electromagnetic interferences. Therefore, the representativeness of fault injection needs to be justified in order to achieve credible results about software fault tolerance. Christmansson and Chillarege [41] proposed a procedure for defining error models such that injected errors emulate software faults and not hardware faults. This property is achieved by defining a mapping between injectable errors and representative software faults, using field failure data about the system under analysis. The procedure is aimed to *fault forecasting* purposes, such as to estimate coverage of software fault tolerance to be used in analytical dependability models. It makes use of the following data for each software fault that has been found in the field and fixed:

- the ODC defect type of the fault;
- the ODC defect trigger of the fault;
- the *error type* caused by the fault;
- the software component in which the fault has been fixed.

Based on these data, the procedure defines *what error types should be injected*, and *where the errors should be injected.* Table 2.12 provides the set of error types used in [41] to describe the procedure, and the joint distribution of fault and errors found in the field. Error types are grouped in *Single address* (i.e., the software fault caused an incorrect address word), *Single Non-Address* (i.e., a non-address word is affected, such as data), *Multiple* (i.e., a combination of single errors, or a data structure is affected), and *Control* (i.e., errors affecting memory in a very subtle and non-deterministic way or not affecting memory at all, such as wrong interaction with a user or terminal communication). It is important to note that many of these error types are specific to the kind of system considered (a mature OS), and their distribution does not necessarily apply to other systems.

Table 2.12: Joint distribution of faults and errors found in the field in the IBM MVS operating system [41].

		ODC defect types					
Error types	**#**	**Chk.**	**Ass.**	**Alg.**	**Tim.**	**Int.**	**Fun.**
Single Address (A)	**78 (19.1%)**	**13**	**27**	**26**	**4**	**5**	**3**
Control block addr.	48	9	16	17	3	2	1
Storage pointer	14	1	5	3	1	2	2
Module addr.	9	3	3	3	0	0	0
Linkage of data structure	4	0	0	3	0	1	0
Register	3	0	3	0	0	0	0
Single Non-Address (N)	**155 (38.0%)**	**30**	**38**	**53**	**12**	**15**	**7**
Value	38	10	6	12	3	2	5
Parameter	38	8	11	10	1	6	2
Flag	37	7	10	17	0	3	0
Length	15	4	4	4	0	3	0
Lock	11	0	1	2	8	0	0
Index	8	1	3	4	0	0	0
Name	8	0	3	4	0	1	0
Multiple (M)	**69 (16.9%)**	**9**	**6**	**32**	**6**	**4**	**12**
Values	4	0	1	2	0	0	1
Parameters	3	0	0	0	0	3	0
Address +	7	1	1	3	0	0	2
Flag +	10	2	1	4	2	0	1
Data structure	37	6	3	20	3	1	4
Random	8	0	0	3	1	0	4
Control error (C)	**106 (26.0%)**	**10**	**7**	**43**	**32**	**5**	**9**
Program management	35	4	0	19	8	2	2
Storage management	33	3	7	12	5	1	5
Serialization	16	0	0	2	14	0	0
Device management	11	2	0	7	2	0	0
User I/O	6	0	0	2	0	2	2
Complex	5	1	0	1	3	0	0
Total	**408**	**62**	**78**	**154**	**54**	**29**	**31**
	100%	**15.2%**	**19.1%**	**37.8%**	**13.2%**	**7.1%**	**7.6%**

The joint distribution of faults and errors is used by the procedure to establish a *connection between the injected errors and software faults they intend to emulate.* The locations where to inject errors (e.g., variables or data structures) are defined by scanning the program code (e.g., using a parser) to identify, for each statement, (i) the ODC defect type that applies to the statement (e.g., the ”Assignment” defect type in the case of an assignment statement), and (ii) the error type that applies to that statement (e.g., the ”Storage pointer” error type in the case that the assignment involves a storage pointer). This processing provides a list of triplets *(location, defect type, error type).* The list is randomly sampled to select where to inject errors, by following the relative frequency of defect and error types in the joint distribution. Moreover, errors should be sampled by taking into account the relative number of faults for each software component, which is also provided by the field failure data (Table 2.13). Finally, a workload representative of field usage is defined on the basis of field failure data on ODC defect triggers (Table 2.14), that is used in fault injection experiments to exercise the target system. As for *when to inject errors*, it is argued in [41] that error injection must be synchronized with the execution of the emulated faulty statement (i.e., an error should be injected every time the location selected for error injection is executed) in order to emulate the permanent nature of software faults. The injection of errors through a SWIFI tool is suggested by the authors, by using a software trap in the code location selected for error injection.

Table 2.13: Fault distribution across components in the IBM MVS operating system [41].

Component	**# of faults**	**# of affected modules**	**Fault / module**
A	43 (10.5%)	33	1.30
B	35 (8.6%)	31	1.13
C	33 (8.1%)	20	1.65
D	32 (7.8%)	28	1.14
E	30 (7.4%)	24	1.25
F	25 (6.1%)	17	1.47
G	25 (6.1%)	23	1.09
H	20 (4.9%)	14	1.43
Others	165 (40.5%)	108	1.53
Total	**408 (100%)**	**298**	**1.37**

Table 2.14: Defect trigger distribution in the IBM MVS operating system [41].

		Error types			
ODC defect trigger	**#**	**A**	**N**	**M**	**C**
Normal mode	284 (69.6%)	47	120	49	68
Startup/Restart	14 (3.4%)	2	1	5	6
Workload/Stress	27 (6.6%)	4	11	3	9
Recovery/Exception	75 (18.4%)	24	19	12	20
HW/SW Configuration	8 (2.0%)	1	4	0	3
Total	**408** **100%**	**78** **19.1%**	**155** **38.0%**	**69** **16.9%**	**106** **26.0%**

An experimental comparison between the injection of representative errors and generic time-triggered errors [42] shown that there can be noticeable differences in the distributions of failure behaviors obtained by these two kind of errors. Therefore, the injection of representative errors should be preferred to obtain more trustworthy results. A procedure similar to [41] has been defined later in [43] for *fault removal* purposes. In this case, errors to be injected are selected by accounting for (i) the failure severity of as perceived by the customers, and (ii) the correlation with fault tolerance deficiencies (i.e., an exception handler or a recovery procedure was active when the fault surfaced in the field, as indicated by Recovery/Exception defect trigger).

Procedures for the injection of representative errors [41, 43] enable the adoption of SWIFI tools for the accurate emulation of software faults. Nevertheless, there are some limitations that prevent the adoption of these procedures in practice. The main limitation is the need for field failure data about the system under analysis. These data are not available in the early phases of the software lifecycle. It is necessary that software has been in usage for long time periods to obtain such data, since failures are rare events. Moreover, field failure data are typically not available for third-party and COTS software. Another issue is that SWIFI tools can accurately inject only a limited number of error types, since error types caused by software faults are in some cases more complex than those caused by hardware faults, as in the case of software faults involving more than one statement, and error types tend to be specific for the system under analysis. These limitations have been faced by approaches for the injection of *actual faults* in the program code, which are discussed in subsection 2.4.3.

2.4.2 Interface error injection

The injection of interface errors is a particular form of error injection that corrupts the *inputs values* provided to a target software component, or the *output values* that the target provides to other software components, to the hardware or to the environment. The injection of errors at input parameters is aimed to emulate the effects produced by faults outside the target, including the effects of software faults in external software components, and to evaluate the ability of the target to detect and handle corrupted inputs. In a similar way, the corruption of output values is adopted to emulate the outputs of faulty components, such as a COTS components, and can be used to assess the impact of faults on the rest of the system.

The corruption of input parameters can reveal deficiencies in the design and implementation of error detection and recovery mechanisms of the target (e.g., input handling code). It is commonly adopted in the context of *robustness testing*, which evaluates "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions" [1]. It should be noted that robustness testing and interface error injection are quite different than functional testing techniques, such as black-box testing: they are concerned with assess the robust behavior of a software module in face of corrupted inputs (e.g., a process crash is avoided, or a warning signal is produced), rather than with functional correctness of the target.

The injection of interface errors at input parameters is especially useful for testing software components that provide a generic *Application Programming Interface* (API). In such components, which are generic and not developed with a specific system in mind, interface errors are likely to arise when they are integrated with other fault-prone software, such as

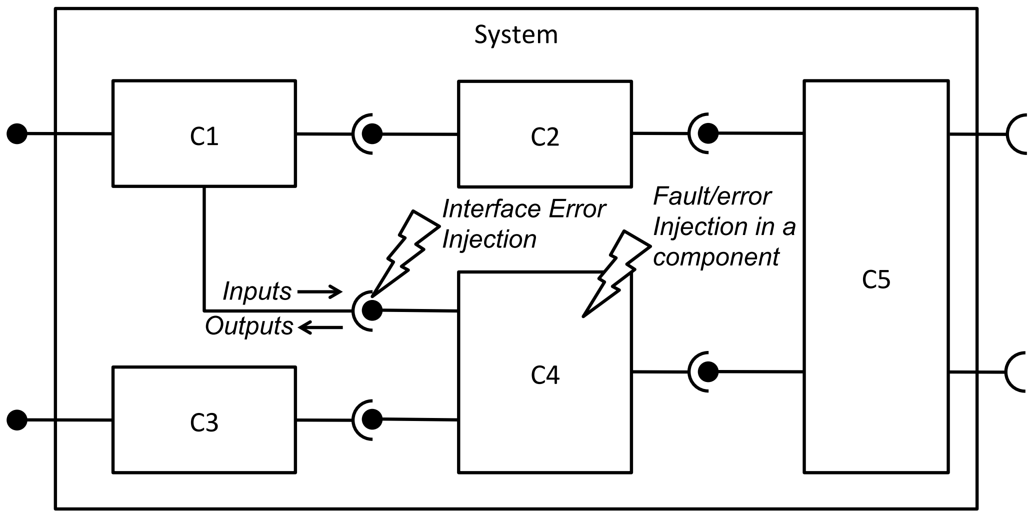
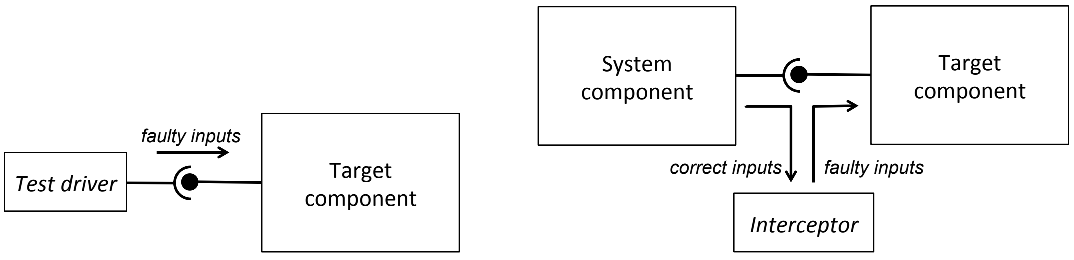

System
C1
C2
C3
C4
C5
Interface Error Injection
Fault/error Injection in a component
Inputs
Outputs

Figure 2.9: Comparison between Interface Error Injection and Fault/Error Injection in a component.

COTS and third-party software, or can be provoked by an incorrect usage of the API. An example is represented by operating systems that provide an API for developing device drivers: drivers are typically provided by third-party suppliers in order to support new devices, and tend to be much more fault-prone than other OS components [39, 80]. Therefore, APIs provide error detection and recovery mechanisms to prevent interface errors from further propagating through the component.

Interface error injection can be performed in two ways (Figure 2.10). The first approach is based on a *test driver* program that is linked to the target component (e.g., a program that uses the API exported by the target), and that generates test cases by submitting invalid inputs. This approach resembles unit testing, where robustness rather than functional correctness is evaluated. The second approach consists in intercepting and corrupting the interactions between the target and the rest of the system, i.e., an *interceptor* program is notified when the target component is being invoked, and it modifies the original inputs in order to introduce a corrupted input. In this scenario, the target component is tested in the context of the whole system that integrates the target. This approach resembles SWIFI, since the original data (in this case, interface inputs) flowing through the system is replaced with corrupted data.



(a) Test driver. (b) Interceptor.

Figure 2.10: Approaches for interface error injection.

In interface error injection experiments, typically only one input parameter and one invocation is corrupted, among the several input parameters and the several invocations of the target API that take place during the experiment. There are three common approaches for generating invalid input values:

1. **Fuzzing**. The original value is replaced with a randomly generated value.
2. **Bit-flipping**. The corrupted value is generated by inverting the value of one or more bits of the original value.

3. **Data-type based injection**. The original value is replaced with an invalid value, which is selected on the basis of the *type* of input parameter being corrupted, where the types are derived from the programming language interface exported by the target. This approach defines a pool of invalid values for each data type, which are selected from the analysis of the type domain.

One of the earliest study on interface error injection has been presented in [141], which evaluated the robustness of UNIX basic utilities in the presence of random inputs (i.e., the occurrence of a process crash or stall). Two tools, respectively *Fuzz* and *ptyjig*, are proposed to submit a random stream of data to the target through the standard input and through the terminal device. The study found that a significant number of utility programs on three UNIX systems (between 24% and 33%) is vulnerable to interface errors. Moreover, error injection was instrumental in pointing out several bugs, such as buffer overruns and unchecked return codes. A subsequent experiment [140] found that the same utilities were still affected by a significant part of faults found in [141] 5 years before, and that similar issues were present in network and graphical applications.

Although the fuzzing approach is simple to implement and can reveal deficiencies in error handling and recovery, its efficiency is questioned by several studies, since it relies on many trials and "luck". This approach has been extended in the Random and Intelligent Data Design Library Environment (*RIDDLE*) [82], that has been designed for robustness testing of Windows NT utilities. To generate erroneous inputs, RIDDLE adopted an approach based on the input grammar of the component under test, described in a form similar to the Backus-Naur form. The grammar is used to generate erroneous inputs (random and boundary values) that are syntactically correct, since unstructured random tests tend to focus on the input handling code of the program.

In order to improve the effectiveness of robustness testing, other studies investigated the data-type based error injection approach, which focuses on invalid values that are typically problematic to handle. In [57], a data-type based approach is proposed to test a Real-Time OS adopted in a fault-tolerant aerospace system. The RTOS is tested against invalid inputs passed to its *system call interface*, in order to assess the ability to handle errors generated by faulty user-space programs. The study considers system calls related to the file system (e.g., create, read, and write files), the memory system (e.g., allocation and deallocation of memory blocks), and the inter-process communication system (e.g., post a message and wait for a message). Each experiment consists of a system call invocation with a combination of both valid and invalid parameters, which are performed by a test driver. For each group of system calls and data type, the study defines a set of input values (e.g., closed or read-only files, and NULL or wrong pointers to memory areas). The test outcome is determined by recording the error code returned by the system call (e.g, the error code reflects or not the invalid input, or an error code is not returned at all), and by monitoring system processes using a watchdog process (e.g., a process unexpectedly terminates during the experiment, or it is stalled). The test campaign found several deficiencies in the target RTOS, some of them leading to severe consequences on the RTOS and tasks other than the test driver (e.g., a cold restart of the whole system is needed).

A general framework for designing benchmarks of software robustness, based on interface error injection, was proposed in [145] in order to allow the comparison of different systems. This work discussed for the first time the aspects to be taken into account for a disciplined comparison of software dependability, which are described in detail in subsection 2.5.2. Moreover, this study proposed a *hierarchical approach* to define the inputs to be injected, in which the different kinds of resources (e.g., log or storage) are abstracted away from their implementation in a specific system or subsystem under test (e.g., a storage object can denote a file or a memory buffer), in order to improve the portability and extensibility of a benchmark.

Subsequent studies on robustness testing and benchmarking of operating systems [125, 123, 55, 124] proposed the *Ballista* approach for testing and comparing operating systems compliant to the POSIX system call interface [3]. Ballista is a highly scalable approach, as only 20 data types had to be defined to test 233 system calls of the POSIX standard. Data types are used to automatically generate a test driver program for each test case. For each data type, Ballista considers a subset of exceptional values from the data type domain.

These values are suggested by the testing literature or selected based on the experience of developers, and represent situations that are likely to be exceptional in some contexts. Examples of three data types used in the *write* system call are provided in Table 2.15 [124]. This system calls takes in input three parameters, namely a file descriptor, a pointer to a memory buffer, and an integer representing the buffer size. Exceptional values for a file descriptor parameter include a file that has opened and delete from the file system, and a file with insufficient access permissions. In a similar way, exceptional values for a memory buffer parameter include extremely large buffers, or a buffer that has been previously deallocated. It is important to note that the definition of data types is not tied to the semantic of a specific system call (e.g., the "memory buffer" data type can be used for both the *write* system calls and other system calls involving memory buffers). A test case for the *write* system call is represented by any combination of values from the three columns of Table 2.15.

Moreover, Ballista provides a *failure severity scale*, which is referred to as *C.R.A.S.H.*, to be used for categorizing test results and compare different systems:

- ***C**atastrophic.* The OS state becomes corrupted or the machine crashes and reboots.
- ***R**estart.* The OS never returns control to the caller of a system call, and the calling process is stalled and needs to be restarted.
- ***A**bort.* The OS terminates a process in an abnormal way.

Table 2.15: Ballista data types for testing the *write* system call [124].

File descriptor	**Memory buffer**	**Size**
FD_CLOSED	BUF_SMALL_1	SIZE_1
FD_OPEN_READ	BUF_MED_PAGESIZE	SIZE_16
FD_OPEN_WRITE	BUF_LARGE_512MB	SIZE_PAGE
FD_DELETED	BUF_XLARGE_1GB	SIZE_PAGEx16
FD_NOEXIST	BUF_HUGE_2GB	SIZE_PAGEx16plus1
FD_EMPTY_FILE	BUF_MAXULONG_SIZE	SIZE_MAXINT
FD_PAST_END	BUF_64K	SIZE_MININT
FD_BEFORE_BEG	BUF_END_MED	SIZE_ZERO
FD_PIPE_IN	BUF_FAR_PAST	SIZE_NEG
FD_PIPE_OUT	BUF_ODD_ADDR	
FD_PIPE_IN_BLOCK	BUF_FREED	
FD_PIPE_OUT_BLOCK	BUF_CODE	
FD_TERM	BUF_16	
FD_SHM_READ	BUF_NULL	
FD_SHM_RW	BUF_NEG_ONE	
FD_MAXINT		
FD_NEG_ONE		

- ***Silent.*** The OS does not return an indication of an error in the presence of exceptional inputs.
- ***Hindering.*** The OS returns an incorrect error code, i.e., the error code reports a misleading exceptional condition.

A robustness testing campaign using Ballista, reported in [124], allowed the comparison of 15 COTS OS. A total of 1,082,541 test cases were automatically executed. Several Abort and Restart failures were observed, along with some Catastrophic failures that affect the system as a whole. The most prevalent sources of robustness failures were illegal pointer values, numeric overflows, and end-of-file overruns. On the one hand, these results are useful for improving exception handling in OS; on the other hand, they highlight the importance of robustness testing for complex software, and in particular for COTS components. The Ballista approach was also adopted in subsequent studies on robustness testing of Microsoft Windows operating systems APIs [178], and of CORBA ORB implementations [157].

The injection of interface errors has also been adopted for testing microkernel-based operating systems by the Microkernel Assessment by Fault injection AnaLysis and Design Aid (*MAFALDA*) [168, 13]. MAFALDA is aimed at injecting both errors in the executable code of a microkernel (in order to emulate hardware errors or software faults in OS code), and interface errors at the API of the microkernel. Interface error injection in MAFALDA is based on the interception of system call invocations during the execution (see also Figure 2.10). MAFALDA has been adopted to analyze the propagation of errors through microkernel components, such as task synchronization, inter-process communication, memory management (see Figure 2.11), and to validate error handling mechanisms in microkernel APIs.

Two techniques are supported by MAFALDA, to be adopted in *trap-based* and *library-based* microkernels respectively (Figure 2.12). In trap-based microkernels, user applications are running in an address space separated from the microkernel. When a system call has to be called, the program issues a *software interrupt* using a special CPU instruction. The CPU execution mode is then switched to supervisor mode, the microkernel handles the interrupt through an interrupt service routine and performs a specific system call. In library-based microkernels, the microkernel is provided as a set of library functions to be linked at

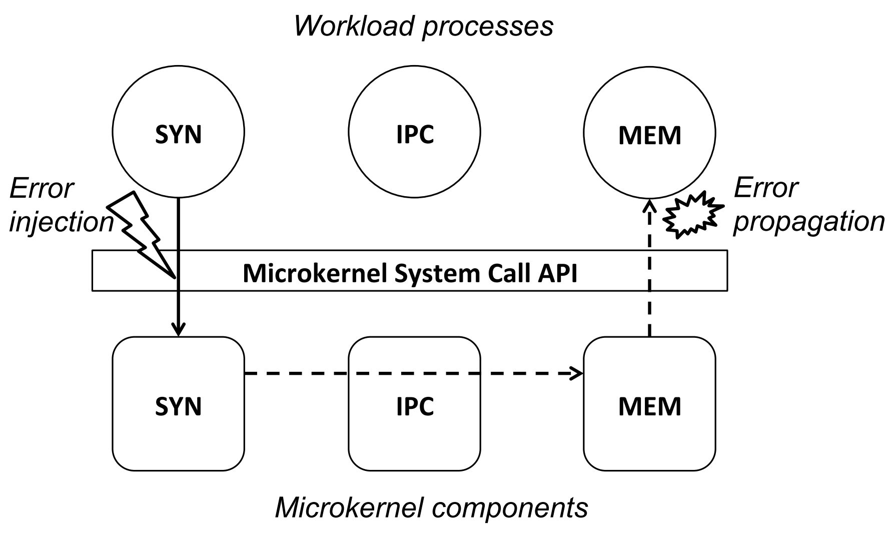
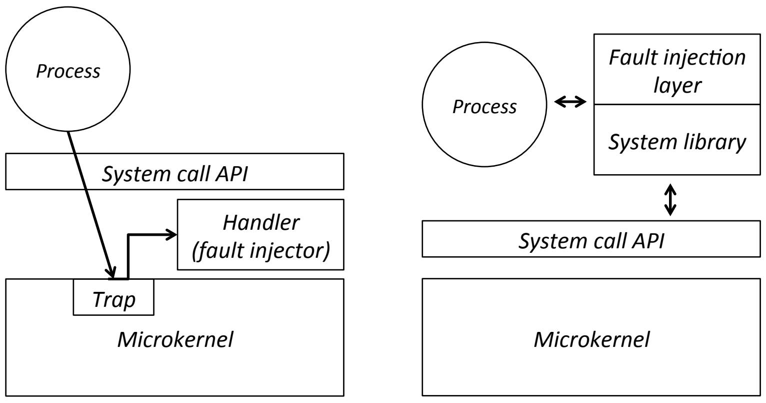


Figure 2.11: Analysis of error propagation in microkernel system using MAFALDA [168].

compile or load time to user software. In some systems, both mechanisms are adopted for implementing system calls, by servicing part of the system call in library code, and another part in supervisor mode. In order to inject interface errors in trap-based microkernels, MAFALDA adopts the software trap mechanism also adopted in SWIFI tools: a trap is set in the interrupt service routine that handles system calls (with the support of hardware debugging features, similarly to Xception), and another trap handler is invoked in order to corrupt an input parameter of the system call. The trap handler then returns the control to the system call interrupt service routine. As for library-based microkernel, interface errors are injected by linking to the user application a fault injection library, which provides the same API of the microkernel library, and which in turn invokes the microkernel library by corrupting input values. Errors are generated through the bit-flipping approach.



(a) Trap-based microkernel. (b) Library-based microkernel.

Figure 2.12: Techniques adopted in MAFALDA to inject faults in the system call interface of microkernels [168].

Other studies focused on the injection of errors at the interface between device drivers and the OS. The extension of the Ballista approach to test the robustness of the device driver interface has been proposed in [7, 108] in the context of Linux and Windows CE, in which input parameters of kernel functions are corrupted using a data-type based approach. They found that OSs are more prone to failures in case of faulty device drivers rather than faulty applications, since developers tend to omit checks in the device driver interface to improve performance, and because they trust device drivers more than applications. Subsequent studies were focused on improving the ability of interface error injection to reveal robustness issues, by carefully selecting the error model to be adopted [110, 205] and the time in which to inject an error [109]. Johansson et al. [110] compared the bit-flipping, fuzzing, and data-type based approaches with respect to their effectiveness in detecting vulnerabilities, and the efforts required to setup and execute experiments. They found that bit-flipping is the more effective approach, although it incurs a high execution cost due to the large number of experiments. Instead, data-type based injection and fuzzing are more efficient, although they incur in a higher implementation cost (e.g., in the case of the data-type based approach, the user has to define exceptional values for each data type). Finally, they found that the best trade-off between effectiveness and cost is obtained by combining fuzzing with selective bit-flipping (i.e., focused on a subset of bits), since the two techniques tend to find different vulnerabilities. In [109], the effectiveness of robustness testing is analyzed with respect to the time in which an error is injected. The approaches typically adopted for interface error injection, namely *first occurrence* (i.e., at the first time in which a code location is executed) and *time-triggered* injection, are compared to a novel approach. The target device driver is profiled in order to analyze the usage profile of functions exported by that driver. The sequence of function calls is divided in subsequences: each subsequence represents a recurring sequence of calls, namely *call blocks*, that denotes a driver state, such as "reading data", or "setting connection parameters". Interface errors are injected at every first occurrence of a function call in each call block. This approach revealed to be useful to improve the effectiveness of robustness testing, since some vulnerabilities are detected only during specific call blocks, and that the workload state has a noticeable impact on the results.

The injection of interface errors has also been adopted for testing the robustness of applications with respect to faults in third-party and COTS software. This goal is achieved by injecting errors in the outputs produced by the target component, in order to assess the ability of the system to handle erroneous values and exceptional conditions. An approach has been proposed in [81] for testing the robustness of software running on Windows NT systems against exceptional conditions (e.g., memory allocation errors, network failures, I/O errors, improper use of OS system calls). This kind of error injection is useful to test error handling code in the program, which is difficult to test and debug with traditional testing approaches. The approach proposed in [81] performs error injection by intercepting and corrupting values exchanged between OS library code and the user program (in the case of Windows NT, these libraries are referred to as *dynamic-link libraries*, DLL). The interception is performed by modifying the Import Address Table (IAT) of the user program executable, which is used at run-time in order to look-up DLLs to be linked to the program. The IAT is modified by replacing the original addresses (that point to DLL functions) with addresses pointing to a *wrapper library*: it is a library with the same API of the original DLL that forwards any function call to the original DLL, and that performs fault injection by returning an error code or exception. A similar approach is adopted by the *FIG* tool [27], which injects error codes in library function calls in UNIX systems, by using a wrapper library that is dynamically linked to the user program. A limitation of these tool is the need for manually specifying error codes to be injected, which are hardcoded in the wrapper library. This operation is time consuming, since a new wrapper library has to be developed in order to injects errors for a different target library, and also error-prone since documentation about the library can be missing or wrong. The Library Fault Injector (*LFI*) [137, 136] mitigates this problem by identifying the error codes of a library by static analysis of library code. The results of static analysis are used to produce *fault injection scenarios*, where each scenario is used to automatically generate a wrapper library for injecting an error. The fault injection scenario specifies the library function which should return an error, the error to be returned, and the number of function calls after that the error is triggered.

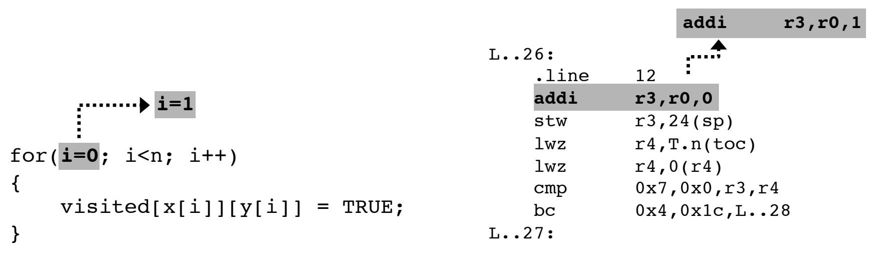
Another approach for injecting interface errors, proposed in [138, 144] and implemented in the *Jaca* tool, is based on *reflective programming*, which is a support provided by modern programming languages that allows a program to inspect and manipulate its own structure and behavior at runtime [134]. Jaca injects interface errors in Java programs, by modifying input and output values of class methods through reflection mechanisms provided by the Java Virtual Machine (JVM), which are used to introduce *class wrappers* that intercept and modify input and output values. Moreover, Jaca avoids the need for source code by performing reflection at the bytecode level, it is designed to be portable across JVM implementations, and it adopts an extensible architecture that allows the introduction of new fault models.

2.4.3 Fault injection through code changes

The studies discussed in previous subsections emulate software faults by the injection of *fault effects* (errors) using SWIFI approaches, although they are limited by the issue that the injected errors (e.g., bit-flips and stuck-at errors) do not necessarily represent errors generated by software faults. To tackle this issue, more recent studies focused on the injection of *actual faults* in the program code, based on the observation that changing the code of the target to introduce a fault is the closest thing to having the fault there in the first place. However, this is not easily achieved as it requires to know exactly where in the target code one might apply such change, and knowing exactly what instructions should be placed in the target program.

The experimental study reported in [133] analyzed whether it is possible to inject faults in a program by applying SWIFI on the code memory area of a process or, equivalently, on the binary executable before running the experiment. To this aim, the study considered a set of programs developed during a programming contest, and that were believed to be correct by the contest judges. The authors identified a set of representative software faults, by thoroughly testing these programs. They then tried to emulate these faults by using the Xception SWIFI tool on the PowerPC 601 hardware architecture.

This experiment highlighted that SWIFI tools can inject software faults only to a limited extent, and that tools and techniques specifically tailored to Software Fault Injection are



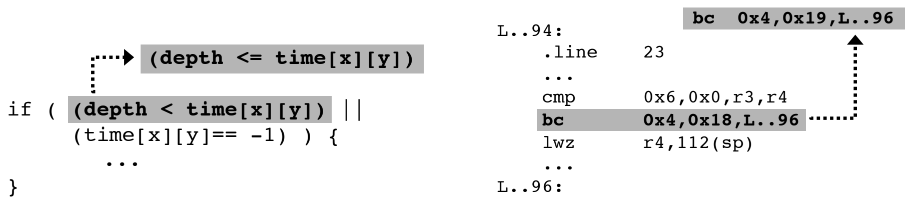
```
i=1

for(i=0; i<n; i++)
{
    visited[x[i]][y[i]] = TRUE;
}
```

(a) Source code.

```
                       addi    r3,r0,1
L..26:
    .line     12
    addi      r3,r0,0
    stw       r3,24(sp)
    lwz       r4,T.n(toc)
    lwz       r4,0(r4)
    cmp       0x7,0x0,r3,r4
    bc        0x4,0x1c,L..28
L..27:
```

(b) Machine code.

Figure 2.13: Emulation of an Assignment fault using SWIFI [133].



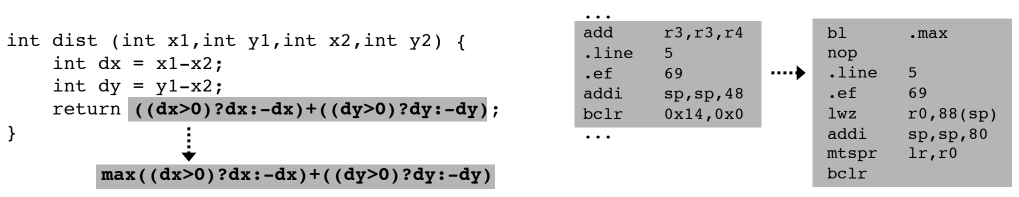
```
      (depth <= time[x][y])

if ( (depth < time[x][y]) ||
     (time[x][y]== -1) ) {
        ...
}
```

(a) Source code.

```
                 bc  0x4,0x19,L..96
L..94:
    .line    23
    ...
    cmp      0x6,0x0,r3,r4
    bc       0x4,0x18,L..96
    lwz      r4,112(sp)
    ...
L..96:
```

(b) Machine code.

Figure 2.14: Emulation of a Checking fault using SWIFI [133].



```
int dist (int x1,int y1,int x2,int y2) {
    int dx = x1-x2;
    int dy = y1-x2;
    return ((dx>0)?dx:-dx)+((dy>0)?dy:-dy);
}
      max((dx>0)?dx:-dx)+((dy>0)?dy:-dy)
```

(a) Source code.

```
...
add      r3,r3,r4          bl       .max
.line    5                 nop
.ef      69                .line    5
addi     sp,sp,48          .ef      69
bclr     0x14,0x0          lwz      r0,88(sp)
...                        addi     sp,sp,80
                           mtspr    lr,r0
                           bclr
```

(b) Machine code.

Figure 2.15: An Algorithm fault that SWIFI cannot emulate [133].

needed. In particular, SWIFI was able to correctly emulate the Checking and Assignment ODC defect types, as the injected code is very close to the intended fault (see the examples in Figure 2.13 and 2.14). However, there were issues in the injection of some Assignment faults when the faulty assignment caused the relocation of many variables in stack memory: to inject this kind of Assignment faults using SWIFI, many software traps or breakpoint registers would be required, therefore increasing the intrusiveness and the hardware support required for injection. Moreover, manual intervention would be needed to identify fault locations and to define the bit-level operations to be performed. While the Assignment and Interface ODC defects types could potentially be emulated by extending SWIFI tools, the study points out that SWIFI cannot inject faults belonging the Algorithm, Function, and Timing ODC defect types. An example of Algorithm fault is presented in Figure 2.15, in which the fault (a missing invocation of the *max* function) affects the translation of several statements, and therefore complex code manipulations and large manual intervention would be required in order to inject this fault through SWIFI.

Another experimental study conducted in [104, 105] compared interface errors, that were injected at the system call interface of the Linux kernel, to errors injected at the interfaces of internal kernel functions, and to real software faults found by static code analyzers. The study compared these faults/errors by (i) the distributions of failure modes, (ii) the return codes of system calls, and (iii) the effects on internal assertions that were introduced in the kernel to monitor its behavior, such as the amount of memory allocations in the kernel. The analysis revealed that external interface errors (i.e., injected at the system call interface) cause a different behavior than internal errors, which are in turn different than real software faults. Therefore, it is concluded that interface errors do not necessarily represent the effects of software faults. The comparative analysis presented in [142] on two complex systems (a real-time OS and an object-oriented DBMS) confirmed the result of [104, 105]. It analyzed interface errors produced by a component when software faults are injected in its code, and compared these errors with those produced by techniques for interface error injection. It observed that injected interface errors and injected software faults produce different errors and failure modes, and they should be regarded as complementary, rather than alternative, means for the assessment of software systems.

The first attempt to inject software faults in a program code for dependability evaluation purposes roots back to [97], in which a program mutation tool (*FAUST*) has been used to assess the effectiveness of several fault-tolerant techniques by introducing defects in the source code of target programs. It should be noted that, although previous studies proposed the injection of faults in the code of a program for defining and evaluating test cases (this application is referred to as *mutation testing* [90, 54, 121]), the suitability of this approach for assessing fault-tolerant systems is unclear. The mutations adopted in [97] targeted the control flow, array boundaries, mathematical expressions, and pre/post increment/decrement operations. Although the adoption of mutations broadens the scope of fault injection to software faults, the representativeness of mutations with respect to real software faults was still a neglected issue, which is required for achieving trustworthy results. Moreover, mutation testing tools usually assume the availability of the source code, which is likely the case since mutation testing is adopted on the software being developed. However, the absence of source code is a limitation for fault injection, since fault injection should be applicable to third-party software for which the source code is not available, such as COTS components. The relationship between fault injection and mutation testing is further discussed in section 2.6.

The Fault Injection and Monitoring Environment (*FINE*) [119] was the first tool specifically aimed to the injection of *actual faults*, by introducing faults in the executable (binary) code of a target program. FINE was developed to analyze the behavior of UNIX operating systems in the presence of hardware and software faults. This tool has been later extended by the *DEFINE* tool [118] to support the execution of experiments in a distributed environment, and included additional fault types related to hardware and communication. The FINE and DEFINE tools have been used to study the impact and propagation of faults respectively in the SunOS UNIX operating system and in the SUN NFS distributed filesystem.

The fault model for software faults that was adopted by FINE (Table 2.16) encompassed

Table 2.16: Fault model for software faults adopted by FINE [119].

Fault type	**Description**
Initialization	The initialized value is replaced with an incorrect value, or no initial value is given (the corresponding instructions are changed to *nop*s)
Assignment	The assignment destination is assigned a wrong value, the evaluated expression is assigned to the wrong destination (and the right destination is not assigned the evaluated value), or the assignment is not executed (the corresponding instructions are changed to *nop*s)
Checking	The branch instruction is replaced with *nop* for a missing condition check, or the condition check is changed for an incorrect condition check
Function	A user-defined sequence of instructions is replaced with another user-defined sequence of instructions (faulty instructions should fit into the space of the original instructions)

a set of defect types that were proposed in an early version of the Orthogonal Defect Classification [38], namely Initialization, Assignment, Checking, Function, and Documentation. The Initialization, Assignment, and Checking types were actually implemented in FINE, while the definition of the Function type (for which there were no special fault patterns) was left to the user. The tool aimed to provide some confidence on the results by covering these fault types that reflect most of the defects found in the field. The faults are injected by changing the executable code of the target, in order to avoid the need for the source code and the costs of mutating and recompiling the source code several times. However, the studies on FINE and DEFINE [119, 118] lacked a precise description of a procedure about how these fault types should be injected, such as how to select an incorrect value or variable to be injected in an incorrect initialization/assignment, how to introduce an incorrect branch instruction, or how Function faults should be injected.

Another approach for injecting software faults was proposed in the context of a work

Table 2.17: Fault model for software faults proposed by Ng and Chen [152, 153].

Fault type	**Example** Correct code	Faulty code
Initialization	`function () { int i=0; ... }`	`function () { int i; ... }`
Missing random instruction	`for(i=0; i<10; i++,j++) { body }`	`for(i=0; i<10; i++) { body }`
Incorrect destination register	`numFreePages = count(freePageHeadPtr)`	`numPages = count(freePageHeadPtr)`
Incorrect source register	`numPages = physicalMemorySize/pageSize`	`numPages = virtualMemorySize/pageSize`
Incorrect branch	`while(flag) { body }`	`while(!flag) { body }`
Corrupt pointer	`ptr = ptr->next->next`	`ptr = ptr->next`
Allocation management	`ptr = malloc(N); use ptr; use ptr; free(ptr);`	`ptr = malloc(N); use ptr; free(ptr); use ptr again;`
Copy overrun	`for(i=0; i<sizeUsed; i++) { a[i] = b[i]; }`	`for(i=0; i<sizeTotal; i++) { a[i] = b[i]; }`
Synchronization	`getWriteLock; write(); freeWriteLock;`	`write();`
Off-by-one	`for(i=0; i<size; i++)`	`for(i=0; i<=size; i++)`
Memory leak	`ptr = malloc(N); use ptr; free(ptr); return;`	`ptr = malloc(N); use ptr; return;`
Interface	`results = strcmp(str1, str2);`	`results = strcmp(str1, str3);`

on the design and verification of a fault-tolerant write-back file cache in the FreeBSD OS [152, 153]. The fault model proposed in that work (Table 2.17) was based on a field failure data study on operating systems that preceded the Orthogonal Defect Classification [187]. Some of these fault types are injected by modifying instructions and their operands in the machine code, in a similar way to FINE (e.g., missing initialization, incorrect source or destination register, incorrect branch, pointer corruption). The remaining fault types are injected by modifying the behavior of kernel functions: (i) in the case of faults in allocation management, the *malloc* kernel function occasionally frees the memory after a delay; (ii) in the case of synchronization and memory leak faults, kernel functions such as *free* do not perform the required operation; (iii) in the case of copy overrun faults, kernel functions for copying data occasionally increase the number of copied bytes, according to the distribution reported in [187]. This approach is an important step towards the injection of representative faults, since its fault model is based on empirical data on real faults in a operating system. The same authors in [151] take into account the likelihood of faults (in quantitative terms) in order to probabilistically estimate coverage factors (see Equation 2.2): in that case, the coverage express the ability of the OS to prevent data corruptions in the case of an OS crash. However, this fault model is not general and do not necessarily apply to other systems. Moreover, some important aspects regarding the implementation of the fault model are overlooked, such as the selection of incorrect operands to be used for replacing correct ones. If not properly implemented, this problem could lead to the injection of meaningless faults. For instance, if a register operand is replaced with another register that does not store a program variable (e.g., a register with a temporary variable introduced by the compiler), then the fault in the machine code does not reflect a fault in the source code.

The problems of defining a representative fault model, and of accurately injecting faults in the machine (binary) code were investigated more in depth in [60, 61, 65], in which the Generic Software Fault Injection Technique (*G-SWFIT*) was proposed. This technique is based on a fault model representative of the most common faults found in the field. The fault model was initially defined based on common C programming bugs from various sources such as programming manuals, best practice tutorials and error reports [60]. The fault model was later improved in [61, 65] through the rigorous analysis of an extensive set of field data on software faults found in open-source and commercial software, which has been described in subsection 2.3.2. A key point of the fault model is that it is *generic*, since it holds for several software systems and it can be adopted in the absence of field data about the specific system under analysis.

G-SWFIT injects faults in the binary code of the target, which is inspected in order to recognize key programming structures at the machine code-level where high-level software faults (i.e., faults in the source code) can be emulated. The technique is based on a *fault library* that defines a set of *machine code patterns* corresponding to programming constructs in the source code, which the technique uses for introducing faults into the binary executable. These patterns are carefully designed by taking into account a specific hardware platform, and assuming that a specific source code compiler has been used to compile the target binary (the patterns are tailored to a specific hardware architecture and compiler). More specifically, the fault library of G-SWFIT provides a set of *fault emulation operators* (Table 2.18). Each fault operator defines:

- **A search pattern**: a sequence of machine code instructions that relate to the high-level constructs where faults can be emulated.

- **A code change**: the mutation to be introduced in a location to emulate of the intended fault type.

Table 2.18: Fault operators of G-SWFIT [65].

Fault operator	**Description**
OMFC	Missing function call
OMVIV	Missing variable initialization using a value
OMVAV	Missing variable assignment using a value
OMVAE	Missing variable assignment with an expression
OMIA	Missing IF construct around statements
OMIFS	Missing IF construct + statements
OMIEB	Missing IF construct + statements + ELSE construct
OMLAC	Missing AND in expression used as branch condition
OMLOC	Missing OR in expression used as branch condition
OMLPA	Missing small and localized part of the algorithm
OWVAV	Wrong value assigned to variable
OWPFV	Wrong variable used in parameter of function call
OWAEP	Wrong arithmetic expression in function call parameter

The proposed fault operators emulate valid faults in terms of programming language, i.e., changed code reproduces high-level faults that are syntactically correct. Fault operators also provide additional rules ("constraints") for selecting fault locations in order to better reproduce the fault types observed in the field. For example, the OMFC fault operator (Table 2.19) only affects function calls that do not return any value or do not make use of the return value (C01), and that are not the only statement in a code block (C02). Constraint C01 is implemented by checking that the CALL instruction is not followed by instructions that represent the usage of the returned value (e.g., by reading the EAX register in the Intel architecture). Constraint C02 is implemented by scanning the binary code around the function call, to find the boundaries of the code block where the call is located. The following instructions mark block boundaries: module entry/exit points, unconditional jumps, and conditional jumps to backward location. Figure 2.16 shows an example of binary code that satisfies C02, but not C01 (the return value is assigned to a variable). Other examples are the OMIFS faults operator (Table 2.20), which removes IF constructs containing no more that 5 statements, the OMLPA fault operator, which removes between 2 and 5 consecutive statements that are not control or loop statements, and the OWPFV fault operator, in which an function parameter is replaced with a variable that must reside in the stack frame and that represents a local variable in the module.

Table 2.19: Description of the OMFC fault operator [65].

Example	*function(...);*
Example with fault	~~*function(...);*~~
Search pattern	*CALL target-address*
Code change	*CALL* instruction removed
Contraints	Return value of the function must not be used (C01) Call must not be the only statement in the block (C02)

G-SWFIT currently represents the state-of-the-art technique for injecting actual software faults in a program, with the respect to both the representativeness of the fault model (based on field data) and the accuracy of faults injected in the binary code (based on a carefully designed fault library). For this reason, G-SWFIT is studied in this thesis in order to evaluate and to improve the representativeness of faults that are injected by this technique.

```
if ( a==123 ) {
    b = function(c);
    c++;
}
```

(a) Source code.

```
mov    off-A[ebp], 123
cmp    off-A[ebp], 0
je     loc-01:

mov    eax,off-C[ebp]
push   eax
call   function-address
add    esp,4
mov    off-B[ebp], eax

mov    ecx,off-C[ebp]
add    ecx,1
mov    off-C[ebp],ecx

loc-01:
```

(b) Machine code.

Figure 2.16: Code blocks in source code and their corresponding machine code [65].

Table 2.20: Description of the OMIFS fault operator [65].

Example	*if(expression) { statements }*
Example with fault	~~*if(expression) { statements }*~~
Search pattern	CMP *reg*, ... JMP *after* ... CMP *reg*, ... JMP *after* *statements* *after:*
Code change	All the conditional jumps to the address *after* are made into unconditional jumps
Contraints	The IF construct must not be the only statement in the block (C02) The IF construct must not be associated to an ELSE construct (C08) *statements* must not include more than 5 statements and not include loops (C09)

2.5 Applications of Software Fault Injection

This section reviews a selection of relevant past studies that adopted Software Fault Injection, in order to highlight its applications and usage scenarios in the assessment and development of fault-tolerant software systems.

2.5.1 Evaluation and improvement of fault tolerance

The main application of fault injection, including Software Fault Injection, is the evaluation and improvement of Fault Tolerance Algorithms and Mechanisms (FTAMs). Examples of FTAMS include N-version programming, recovery blocks, and N-self-checking programming that mask software fault through design diversity [15, 132], and concurrent error detection, checkpointing/recovery, and exception handling that detect an error state and switch to a degraded mode of service [83, 84, 49]. Fault injection aims to validate FTAMs with respect to faults that will be experienced by the system during operation, in order to obtain confidence that the system will be able to deliver a proper service. One means to achieve this confidence is *fault removal* in the design and implementation of FTAMs, that is, detection and removal of deficiencies that cause an incorrect behavior of FTAMs when they are faced with the faults they are intended to handle. Another important means is *fault forecasting*, which aims to rate the efficiency of the operational behavior of the FTAMs, by estimating the parameters that characterize FTAMs, such as coverage factors and latencies.

An early study on rating the effectiveness of software fault tolerance [97] evaluated and compared several fault tolerance techniques through the injection of hardware and software faults. Several teams implemented the same target application following the same specifications. Moreover, each team implemented, along with the software, a different fault tolerance technique, namely N-version programming, recovery blocks, concurrent error detection, and algorithmic fault tolerance. The different implementations were executed using several test vectors and injecting a fault at each execution. In order to identify the outcomes of the experiments (Figure 2.17), the outputs and the internal states are compared to the ones produced by a *golden standard* (i.e., a fault-free reference implementation that was thoroughly tested before experiments). From the analysis of the behavior of the software in the presence of faults, the following probabilities are evaluated:

- **Coverage**: $Pr\{\text{detecting an error} \mid \text{a fault is active}\}$,
- **Recovery**: $Pr\{\text{recovering successfully after error detection}\}$,
- **Aborting**: $Pr\{\text{aborting} \mid \text{successful recovery is not possible}\}$.

From these probabilities, other figures of merit can be derived, such as the probability of correct execution and the probability that the software will behave in a fail-stop manner, that can be used to compare different fault tolerance techniques in terms of reliability and cost effectiveness.

In [119], the injection of both hardware and software faults has been adopted to analyze the propagation of faults in the SunOS operating system and their impact on performance. For this purpose, a Markov model is defined (Figure 2.18), and its transition probabilities and mean state holding times are populated with experimental results. Each state is assigned a *reward*, that is, a numeric evaluation of the performance level provided by the system in that

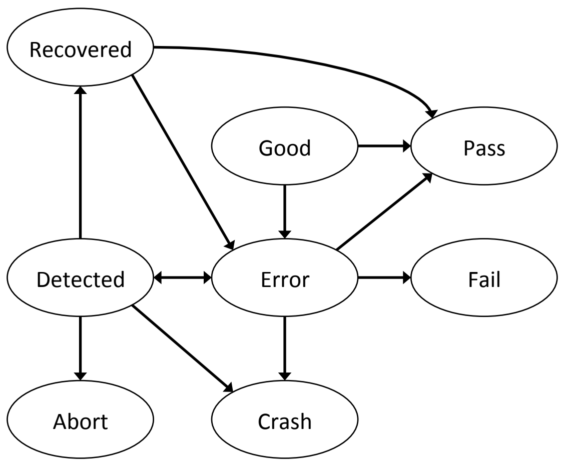


Figure 2.17: Generalized Discrete Markov Model adopted for evaluating and comparing four software fault tolerance techniques [97].

state. A transient Markov reward analysis is performed in order to evaluate the expected performance level in the presence of faults. Bondavalli et al. [25] reported a performability analysis (i.e., evaluation of joint metrics that account for both performance and availability) of a fault-tolerant architecture based on legacy and off-the-shelf software. The architecture is modeled using Stochastic Activity Networks (SAN) [174], whose parameters are populated through fault injection experiments. The model is then analyzed to assess the effectiveness of fault tolerance (in terms of long-term performability of the overall system) and for tuning system parameters (e.g., thresholds used to detect failures).

The estimation of reliability measures obtained by Software Fault Injection can provide useful feedback to the development process for improving the design of fault-tolerant software. An example is provided in [152, 153], in which a write-back file cache (i.e., data are

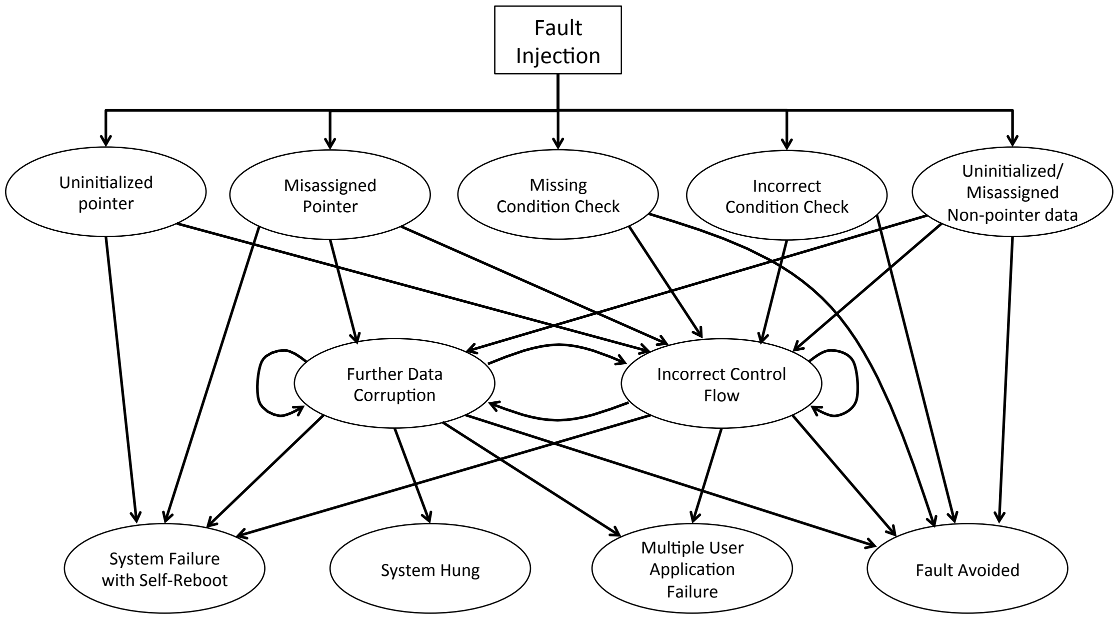


Figure 2.18: Fault propagation models of the SunOS operating system [119].

flushed to the disk asynchronously) is designed with the requirement to be as reliable as a write-through file cache (i.e., data are synchronously written to the disk) in spite of OS crashes. To this aim, an iterative design process is adopted. At each iteration, software faults are injected in the OS in order to induce OS crashes and measure the reliability of the file cache in terms of number of data corruptions caused by the OS crash (Table 2.21). If the file cache is still less reliable than a write-through file cache, then the system behavior under failure is analyzed in detail and the design of the file cache is revised in order to tolerate more crashes. Another example of reliability measure is the percentage of *fail-stop* violations, that is, the system does not immediately halt in case of a failure and exhibits an erratic behavior (e.g., it writes erroneous data to stable storage or sends incorrect information to other systems). The absence of fail-stop violations is an underlying assumption of several fault-tolerant techniques [185, 111, 175]. The validity of this property has been evaluated in a DBMS through Software Fault Injection in [33], where it is observed that the percentage of fail-stop violations (7%) can be reduced by a factor of 3 (2%) using the transaction mechanism to undo recent changes before a crash.

Table 2.21: Corruption rate of file cache designs [153].

Fault type	**Write-Through File Cache**	**Write-Back File Caches**			
		Default FreeBSD Sync	*Basic Safe Sync*	*Enhanced Safe Sync*	*BIOS Safe Sync*
Bit-flips text area	3	51	7	5	2
Bit-flips heap area	0	3	3	2	0
Bit-flips stack area	5	28	8	3	1
Initialization	10	45	9	7	4
Missing random instruction	4	43	8	2	4
Incorrect destination register	4	42	9	5	2
Incorrect source register	4	43	10	3	1
Incorrect branch	4	51	14	4	5
Corrupt pointer	3	38	5	4	2
Allocation management	0	100	5	0	0
Copy overrun	4	36	1	3	2
Synchronization	0	3	1	0	0
Off-by-one	4	59	16	9	3
Memory leak	0	0	0	0	0
Interface	1	47	8	3	2
Corruption rate	46/1500 (3.1%)	589/1500 (39.3%)	104/1500 (6.9%)	50/1500 (3.3%)	28/1500 (1.9%)
95% Confidence Interval	2.2-3.9%	36.8-41.8%	5.6-8.2%	2.4-4.3%	1.2-2.6%

The applications of Software Fault Injection previously described highlight the need for a representative fault model of software faults: if the injected faults do not reflect the real faults affecting the system, then the reliability estimates can be misleading. In [151], the corruption rate of the file cache due to OS crashes is evaluated by taking into account the likelihood of injected faults. The probability of occurrence and activation of each fault type has been derived from field failure data studies on the MVS [187] and Tandem NonStop-UX [189] operating systems (Table 2.22). These probabilities have influence on the estimated corruption rate by accounting for the weight of each fault type (see also Equation 2.2).

Table 2.22: Proportional weighting of fault types [151].

Fault type	**Equal weights**	**MVS**	**Tandem NonStop-UX**
Bit-flips text area	7.69%	0.7%	6.2%
Bit-flips heap area	7.69%	1.4%	
Bit-flips stack area	7.69%	1.4%	
Incorrect destination register	7.69%	2.7%	6.2%
Incorrect source register	7.69%	2.7%	6.2%
Incorrect branch	7.69%	0.7%	26.0%
Missing random instruction	7.69%	0.7%	6.2%
Initialization	7.69%	5.4%	4.0%
Corrupt pointer	7.69%	14.9%	28.0%
Allocation management	7.69%	2.7%	11.0%
Copy overrun	7.69%	2.7%	
Off-by-one	7.69%	0.7%	6.2%
Synchronization	7.69%	63.5%	
Disk corruption rate	1.08%±0.87%	0.18%±0.17%	1.25%±1.29%
Memory corruption rate	1.54%±1.03%	0.60%±0.68%	1.46%±1.71%

Other works pursued the assessment and improvement of fault tolerance through the injection of faults effects (rather than actual faults in the code). On the one hand, the results of error injection do not translate in probabilistic reliability measures since the representativeness of injected error is difficult to assert. On the other hand, error injection is useful to force "corner cases" in the software that are not easily produced by actual faults in the code, and that can point out weaknesses in the software. An application of error injection is represented by the Extended Propagation Analysis (*EPA*) technique [203]. EPA uses error injection to assess the possibility of unsafe or unacceptable outcomes, and to point out where the software can be extended for preventing error propagation. Given a statement in the source code, the EPA inject errors in the data produced by that statement, in order to evaluate the sensitivity of software to faults in that location. A location producing software failures is a candidate for an assertion that prevents the occurrence or propagation of wrong data states at that location.

A case study on the application of EPA involved a software that controls a device for performing human neurosurgery [203]. The software adjusts the current level in a coil in order to move a seed in a specific location in the brain. The control software communicates with the device by reading and setting coil parameters, and must assure that the current level falls within a specified range in order to prevent patient injury. By using error injection, EPA computes a *failure tolerance* score of a source code location (i.e., percentage of error injection experiments leading to an acceptable outcome). The score of a function or file is defined as the lowest score of locations in that function or file. Table 2.23 shows the failure tolerance score of a subset of locations in the target control software. A sharp decrease in the failure tolerance score can be noticed at line 90. After the inspection of that code, it was found that the variable representing the current value is checked by an assertion before line 90, but the variable is processed (including by a function call to *amps_ to_ dac*) and sent to the hardware without an additional assertion to check its validity. This problem was fixed by moving the assertion, such that it occurs after all value transformations have been made.

Table 2.23: Analysis of failure tolerance in a safety-critical software for medical applications [203].

Source code location	**Failure tolerance score**
test_servo1.sfr	0.48008386
coil.cc	1
basic_functions.c	1
ramp_mdl.cc	1
servoamp.cc	0.48008386
servoamp::servoamp	1
servoamp:: servoamp	1
servoamp::set_current	0.48008386
Line 89	1
Line 90	**0.48008386**
Line 94	0.83857442
Line 95	1
Line 96	0.95387841
servoamp::get_actual_current	1
servoamp::get_current_settings	1
servoamp::read_fault	1
servoamp::inhibit	1
servoamp::uninhibit	1
servoamp::get_amp_status	1
servoamp::read_config_file	1
servoamp::adc_to_amps	1
servoamp::amps_to_adc	1
servoamp::amps_to_dac	0.83857442
Line 350	0.83857442
servoamp::dac_to_amps	1

The Propagation Analysis Environment (*PROPANE*) [93, 94, 191] has been later proposed for the analysis of error propagation in component-based software. PROPANE injects errors using SWIFI at the inputs of a component (e.g., by corrupting messages or function parameters), and logs its outputs in order to identify an error propagation by comparing outputs to a golden run. Experimental results are used to compute the *error permeability* of components, that in turn can be used to identify vulnerable or critical components that deserve more attention during design (e.g., for selecting suitable locations for error detection and recovery mechanisms).

The injection of errors at component interfaces has also been exploited for designing protective *wrappers*, i.e., additional software layers interposed between a component and the rest of the system. Error injection can be used to identify inputs that are not gracefully handled by the component, which can be rejected or handled by introducing *input wrappers* (Figure 2.19), or component outputs that are not tolerated by the system, which are filtered by *output wrappers* [202]. In [172, 168], a COTS microkernel was extended with protective wrappers, in order to prevent error propagation through the microkernel. In this case, wrappers are manually derived from specifications of its functional classes (e.g., synchronization, scheduling, memory) and validated through error injection. *Xept* [200] is a tool for generating wrapper libraries that perform exception handling, which are specified by the user using a C-like language.

An automated approach for deriving protective wrappers (*HEALERS*) was proposed in [74, 73] for C and C++ libraries, in which error injection experiments (using the BALLISTA approach) are performed to identify inputs handled in a robust manner, and to automatically generate a library wrapper that serves as input filter. In a similar way, the approach proposed in [186, 188] (*AutoPatch*) aims to enhance the robustness of applications with respect to *error codes* returned by external libraries, by injecting error codes in order to find and to patch bad error handling in application code. Another approach for improving error handling has been proposed in [72], in which *exceptions* are injected in C++ and Java program in order to identify exception handlers that leave an object in an inconsistent state (e.g., a file or socket handler opened by a method are left opened if an exception occurs).

2.5.2 Dependability Benchmarking

An important field of application for fault injection, which has been developed in the last decade [101, 117], is represented by *dependability benchmarking.* A dependability benchmark is a means to characterize and to compare the dependability of a computer component or system in the presence of faults. A key aspect of dependability benchmarking, which makes it different from existing dependability evaluation techniques, is that it represents an *agreement* that is widely accepted both by the computer industry and by the user community. The technical agreement states the measures, the way and conditions under which the measures are obtained, and the domain in which these measures are considered valid and meaningful.

```
char* asctime (const struct tm* a1) {
        char* ret;

        if (in_flag) {
                return (*libc_asctime)(a1);
        }

        in_flag = 1;

        if (!check_R_ARRAY_NULL(a1,44)) {
                errno = EINVAL;
                ret = (char*) NULL;
                goto PostProcessing;
        }

        ret = (*libc_asctime)(a1);

        PostProcessing:;
        in_flag = 0;
        return ret;
}
```

Figure 2.19: Wrapper code for the *asctime* UNIX function [74]. The wrapper checks that the *a1* parameter is a string pointer to correctly allocated memory, and returns an error if this is not the case.

This agreement implies that the benchmark should specify in detail the procedures and rules to be followed in order to enable users to implement the benchmark for a given system, and to interpret the benchmark results.

A general framework for dependability benchmarking has been defined in the context of the DBench European project [101]. This effort was followed by several studies that defined and refined dependability benchmarks for many kind of systems (e.g., OLTP systems, general purpose and real-time operating systems, engine control applications), and resulted in the recent publication of a book [117] on this topic. The general framework of a dependability benchmark identifies three main dimensions that have to be considered in a dependability benchmark:

- **Categorization**. This dimension describes the benchmarked system (*Benchmark Target*, BT) and the context of the benchmark, which determine the requirements and the objectives of the benchmark and are used to define realistic experimental conditions. It includes the *domain* in which the system is adopted, the *operating environment* in which the system will work, the *life cycle phase* of the BT in which the benchmark is executed, the *benchmark performer* and the *benchmark user*, and the *benchmark purpose* (e.g., comparing systems, tuning system configuration, improving fault tolerance).
- **Measure**. This dimension defines the measures that are relevant for the dependability benchmark. It specifies whether measures are *qualitative* or *quantivative*, whether they are *dependability* or *performance* related (e.g., to assess performance degradation in the presence of faults), whether they are *comprehensive* (i.e., measures of interest for the end-user that characterize system at the service delivery level, such as the number of transactions per minute) or *specific* (i.e., associated to a particular feature of the system, such as coverage and latency of fault tolerance mechanisms).

- **Experimentation**. This dimension describes the aspects related to the experiments to be performed on the BT. It includes the *faultload* and the *workload* (which should be representative of faults and of operational profile that are experienced by the BT in its operating environment), and the *measurements* (i.e., readouts) to be collected. Moreover, this dimension should clearly identify and specify the *System Under Benchmark* (SUB), that is, the wider system that is necessary to execute the BT, and to perform the series of experiments defined by the benchmark.

The relationship between the SUB and the BT is clarified in Figure 2.20. The SUB provides the hardware and software support and resources to execute the BT. Moreover, the SUB is also used to apply the workload and faultload, and to collect measurements. In the case of benchmarking of software systems through Software Fault Injection, a fundamental aspect is the clear separation between the BT and the Fault Injection Target (FIT), i.e., the component subject to the injection of faults. This separation is required to avoid the problem of changing the BT, as the injection of software faults introduces small changes or perturbations in a component. The BT should not be modified directly by the faultload in order to assure the credibility of the dependability benchmark, especially from the point of view of the provider of the BT.

It is important to note that fault injection is an important aspect of dependability benchmarking, but it is not the only dependability assessment approach that can be involved.

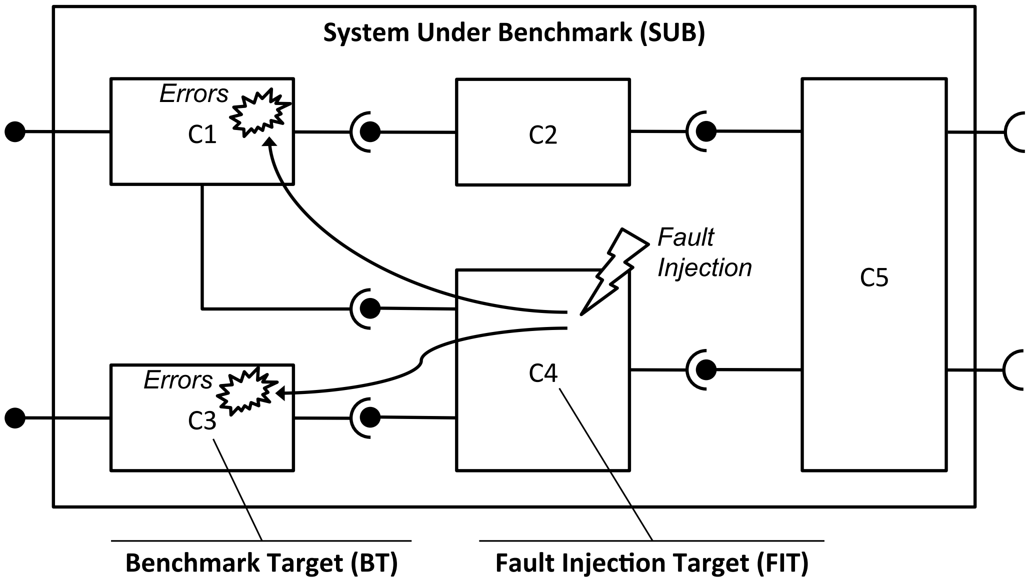


Figure 2.20: System Under Benchmark (SUB), Benchmark Target (BT), and Fault Injection Target (FIT).

In the general case, benchmark measures are obtained from the combination of experimental results (e.g., fault injection) and of the analysis of models (Figure 2.21). An example of measure that is obtained through modelling is the *steady-state availability* of the system, where parameters in the model, such as the coverage of fault tolerance, can in turn be obtained from experiments.

An early study on the use of Software Fault Injection for dependability benchmarking of operating systems appeared in [59, 63]. This study compared three COTS operating systems, namely Windows NT4, Windows 2000, and Windows XP (which represent the BT) with respect to software faults in device drivers (which represent the FIT). Software faults are injected in device driver code by using G-SWFIT. This dependability benchmark

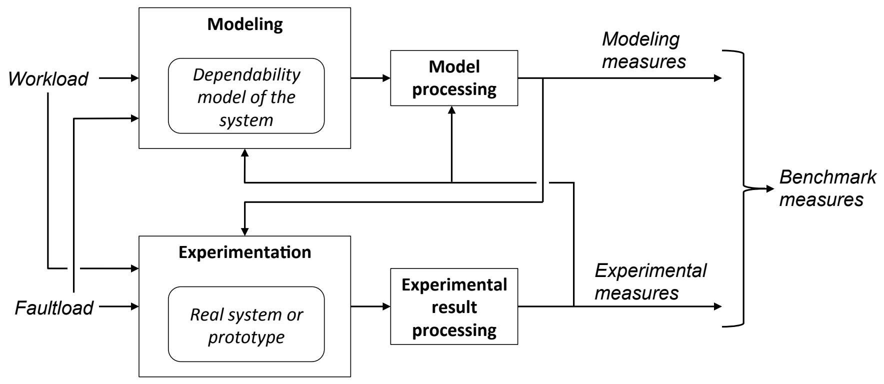


Figure 2.21: Reference model of dependability benchmarking [117].

ranks the target operating systems with respect to the failure modes they exhibit. In order to rank the severity of failure modes, i.e., an ordering of the failure modes from the "worst" to the "better", the benchmark defines three *perspectives* (Table 2.24), namely:

- *Availability*: the failure modes that cause the unavailability of system functionalities or of the whole system are considered the worst ones;
- *Feedback*: the failure modes that provide few or no information to the user about the system state are considered the worst ones;
- *Stability*: the failure modes in which the system is working but provides an incorrect service are considered the worst ones.

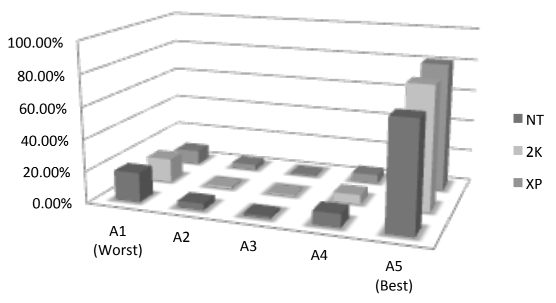
Figure 2.22 provides the distribution of failure modes according to the three perspectives of Table 2.24. A low number of severe failure modes denotes the ability of the operating system to gracefully react to faulty device drivers. The Windows XP exhibited the best

Table 2.24: Classification of failure modes in Microsoft operating systems with respect to faulty device drivers [59].

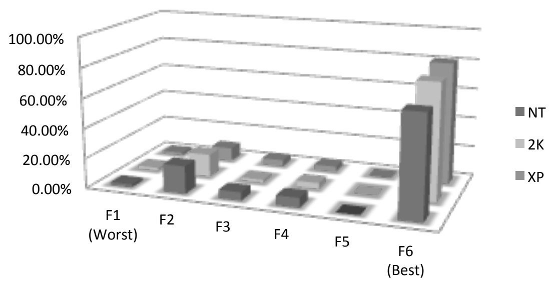
Availability	Feedback	Stability
A1 The machine is completely unusable (not available)	F1 There is no warning and data is lost	S1 There is no warning and data is lost
A2 The machine is available if use of the faulty device is not attempted.	F2 The systems fails without giving any clue to its behavior	S2 The system seems to be in order, but it is not, and it may cause data loss
A3 Sub-systems interacting with the faulty device become unavailable	F3 The drive is not identified by the systems, however the instant when the system or application fails may help its identification	S3 As S2, but with less side-effects (only parts of the systems are affected)
A4 The machine is usable except for the faulty device	F4 Although some clue is given to the user, it is up to him to realize that a device is not working properly	S4 The system refuses to proceed, preventing further damage
A5 The entire system is available	F5 The faulty driver is identified only when first used	S5 The system avoids usage of the malfunctioning part
	F6 The faulty driver has no effect on the OS or is identified, allowing for quick corrective measures	S6 The system behaves normally according to the observations made

behavior for all the three perspectives. The same approach can be adopted to compare the target systems with respect to other perspectives, such as performance and safety. In [7], a similar experiment is performed on the Linux kernel, by injecting data-type based errors at the interface between device drivers and the Linux kernel.

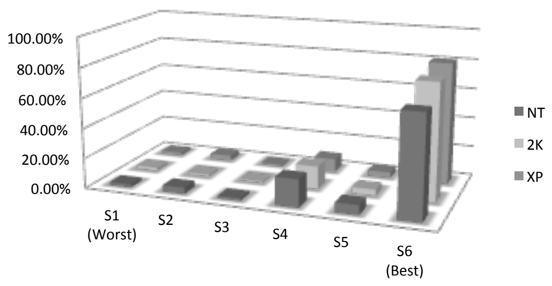
In [112, 116], the concept of dependability benchmarking was further refined, by carefully taking into account the role of the workload. The Windows NT, Windows 2000 and Windows XP operating systems were benchmarked in the presence of faults in user applications. These systems were compared with respect to both their failure modes and their performance in the presence of faults, in terms of their *reaction time* (i.e., time to execute a system call) and



(a) *Availability* perspective.



(b) *Feedback* perspective.



(c) *Stability* perspective.

Figure 2.22: Comparison between the Windows NT, Windows 2000 and Windows XP operating systems [59].

restart time (i.e., duration of OS restart). Software faults are emulated by interface error injection targeted at corrupting the input parameters of system calls. In order to perform error injection in the context of a realistic workload, errors are injected by intercepting system call invocations performed by a realistic workload rather than by using a synthetic test driver. Moreover, workload representativeness is an important aspect since the workload affects the propagation of errors and the reaction of the system in terms of performance and failure modes. Therefore, the definition of a representative workload is instrumental to extend the validity of results to real-world systems. The first definition of the dependability benchmark [112] used as workload the TPC-C performance benchmark for transactional systems [48], in order to take advantage of an already established and agreed workload. The benchmark has later been extended by considering the PostMark workload [120], which is representative of large Internet electronic mail servers, and included Linux-based operating systems among the benchmarked systems.

The first proposal for a complete dependability benchmark based on the injection of actual software faults appeared in [62, 64]. This dependability benchmark, namely *Web-DB*, aims to evaluate and compare web servers with respect to dependability and performance measures (Table 2.25). The faultload is composed by software faults in the operating system code, which are emulated using G-SWFIT in order to achieve fault representativeness, and by operator faults affecting network equipment, and process, connectivity and server management. The workload is represented by the SPECweb99 performance benchmark for web servers [46]. In a similar way to SPECweb99, the Web-DB benchmark includes an initial ramp-up phase, in which the system is exercised to reach its maximum processing throughput, a sequence of injection slots in which one fault at time is injected, and a ramp-down phase in which the workload is terminated. This procedure is generic and has also been adopted to benchmark OLTP systems [198, 197, 117].

Table 2.26 provides the results of Web-DB for the Apache and Abyss web servers, in three

Table 2.25: Dependability and performance measures of the Web-DB dependability benchmark [64].

Measure	**Description**
SPEC	Number of simultaneous conforming connections as defined in the SPECweb99 benchmark (i.e., an average bit rate of at least 320kbps and less than 1% of errors)
Throughput (THR)	Number of operations per second
Responsivity (RTM)	Average response time
Autonomy (AUT)	Percentage of cases in which an administration intervention would not be needed to restore the service (100˘(No. interventions/No. faults) · 100)
Accuracy (ACR)	Error rate (100˘(No. requests with errors/No. requests) · 100)
Availability (AVL)	The percentage of time in which the system is available to execute the workload (Amount of time the system is available during a run/the total duration of that run)

different operating system environments. Results obtained from the injection of software faults and operator faults are averaged and compared to *baseline* results obtained in fault-free experiments. There is not single configuration that is better to the others with respect to every measure. Therefore, having several measures is useful to evaluate the systems with respect to the aspects most relevant to the user, and it is possible to weight these measures in order to reflect the importance of each of them. Nevertheless, it can be noted that the Apache web server provides better results with respect to 5 out of 6 measures (underlined in the table).

2.5.3 Other emerging applications

This section comments on recent and ongoing work on new forms of Software Fault Injection, and new applications differing from the traditional assessment of fault tolerance. A novel form is represented by the integration of Software Fault Injection and *formal methods* in

Table 2.26: Comparison between the Apache and Abyss web servers [64]. Best results for each measure are underlined. Times are expressed in milliseconds.

		Apache						**Abyss**					
		SPEC	**THR**	**RTM**	**AUT**	**ACR**	**AVL**	**SPEC**	**THR**	**RTM**	**AUT**	**ACR**	**AVL**
Windows 2000	Baseline	31	90	345,9	100	100	100	28	82,7	344,4	100	100	100
	Software faults	10,64	83,65	362,18	92,63	94,63	96,72	4,97	75,96	359,69	90,7	90,07	95,61
	Operator faults	17	74,83	402,23	95,32	99,79	93,84	15,67	75,96	367,75	98,02	99,48	97,09
	Average	**13,82**	**79,24**	**382,2**	**93,98**	**97,21**	**95,28**	**10,32**	**75,96**	**363,7**	**94,36**	**94,78**	**96,35**
Windows XP	Baseline	26	74,5	348,9	100	100	100	25	73,3	343,4	100	100	100
	Software faults	13,8	71,67	357,12	93,68	95,56	97,84	10,74	68,69	356,68	93,51	89,42	96,62
	Operator faults	22,33	71,59	362,33	97,28	99,63	98,04	16,67	67,74	367,4	98,43	99,57	98
	Average	**18,07**	**71,63**	**359,7**	**95,48**	**97,6**	**97,94**	**13,71**	**68,22**	**362**	**95,97**	**94,5**	**97,31**
Windows 2003	Baseline	30	82,4	363,9	100	100	100	24	70	345,8	100	100	100
	Software faults	13,79	78,82	371,48	94,72	95,5	97,43	10,4	66,09	355,08	93,55	91,49	96,69
	Operator faults	8,75	79,59	374,77	98,81	99,07	97,81	15,42	66,26	362,27	98,94	99,6	98,37
	Average	**11,27**	**79,21**	**373,1**	**96,77**	**97,29**	**97,62**	**12,91**	**66,18**	**358,7**	**96,25**	**95,55**	**97,53**

the Symbolic Program-Level Fault Injection and Error Detection Framework (*SymPLFIED*) [159]. This framework uses symbolic execution and model checking to verify error detectors in a program, and to expose error cases (single and multiple bit-flips) that would potentially escape detection and cause a failure. SymPLFIED explores the states that can be assumed by a program, and evaluates whether a transient error occurring in each state can evade detection mechanisms and lead to an incorrect result. In this way, SymPLFIED can *prove* the effectiveness of error detection, and it can point out non-detected errors in order to aid in the design of error detection. This approach (and in general symbolic execution) tends to be computationally costly since it requires the exploration of a large set of states, although it is affordable for verifying small programs and algorithms. Moreover, this approach can reveal corner cases that are not found by traditional fault injection due to its inherent statistical nature.

Another novel form of Software Fault Injection is concerned with the emulation of faults in *software requirements* [196]. Although requirement faults do not directly affect software artifacts, it is recognized that wrong requirements (e.g., incomplete, conflicting, incorrect) have a strong impact on software safety, and that they may cause severe accidents [127]. Moreover, is is much more costly and difficult to mitigate these faults during the late phases of the software lifecycle [23]. The emulation of faults in *requirement documents* consists in introducing, removing or modifying requirements, in order to emulate the presence of incomplete, contradicting, or incorrect requirements. This procedure can be useful to provide feedback to the *requirement review process*: the effectiveness of reviews can be evaluated by asking reviewers to inspect a document with injected faults. The field data study of requirement faults in space software in [196] provides a basis to the definition of realistic fault types and of an approach for the automated emulation of these faults.

The use of Software Fault Injection in the context of *software security* has been recently investigated. In this context, SFI emulates *security vulnerabilities* existing in web applications, that is, software defects that may cause unauthorized accesses to confidential data rather than affecting software correctness [16]. The field data study in [75] analyzed the types of software defect that are related to two common software vulnerabilities in web applications, namely *SQL injection* and *Cross Site Scripting* vulnerabilities, and found that most of these software defects are represented by the missing validation and sanitization of input variables, corresponding to an extended version of the MFC fault type of G-SWFIT (Table 2.18). This finding has been exploited to inject vulnerabilities in web applications, in order to train security assurance teams that are responsible for code inspection and penetration testing [76], and to evaluate the effectiveness of intrusion detection systems and vulnerability scanners [77].

Finally, Software Fault Injection is currently being investigated in the context of *online failure prediction* in complex software systems. Online failure prediction aims to *anticipate during runtime* the occurrence of failures in the *near term future*, in order to prevent potential accidents and to limit the impact of failures, and can be seen as a *proactive* form of fault tolerance [171, 170]. Prediction is achieved by monitoring and analyzing the current system state in order to spot *symptoms* that a failure is likely to occur soon, such as an anomalous sequence of events or consumption of system resources. Due to the complexity of systems and to the random nature of failures, failure prediction requires the use of *heuristic rules* or *statistical models*, that have to be trained and validated using symptom and failure data. Unfortunately, this kind of data is typically scarce, since failures are rare events and must be collected over a long time. In [199, 100], Software Fault Injection is indicated as a promising approach for accelerating the data collection process by generating realistic failure occurrences. This data can be exploited to train and to evaluate prediction algorithms in a limited amount of time.

2.6 Relationship with Mutation Testing

The injection of software faults consists of the introduction of small changes in the target program code, creating different versions of a program (each version has one injected software fault). The way faults are injected resembles the well-known mutation testing technique [90, 54, 121] but the injection of software faults has completely different goals. While mutation testing has been used in software verification to identify the best sets of test cases, the injection of software faults is meant to validate fault-handling mechanisms at runtime and to evaluate the way a system behaves in the presence of the injected faults [10, 201, 41, 65]. This difference of goals reflects on the approaches and fault models adopted by SFI.

Mutation testing is a technique for software quality improvement used during the software development phase. The main goal is to improve the ability of test cases to detect faults while maintaining testing time as low as possible [90, 54, 121]. This approach evaluates the effectiveness of test cases (namely, the mutation adequacy score) by executing tests with versions (*mutants*) of the program containing a code mutation (i.e., copies of the original program each containing a small faulty change). These mutations, or faults, are hand-seeded or generated by a set of mutation operators (i.e., rules followed for introducing changes in the code). Test cases are then defined such that they detect as many of the injected faults as possible. The effectiveness of test cases is evaluated by measuring the ratio of mutants that have been *killed* (i.e., the output of the mutant differs from the original program for at least one test case). This approach is based on the assumption that test cases effective against mutants are also effective to detect real faults (this is referred to as the *coupling hypothesis* [54]). Empirical studies confirmed that mutants are suitable for estimating the fault detection ability of test cases, and that automatically-generated mutants are an accurate and more practical support compared to hand-seeded (i.e., manually inserted) faults [9, 58].

There are issues that make this approach costly, and that have been investigated since its birth (a thorough survey is presented in [106]). The foremost issue is the large number of experiments required to run each test case on each mutant. This is due to the large number of mutants that can be generated from a program, since mutation operators encompass many language constructs that can be potentially affected by defects (e.g., "constant replacement" [121]).

It has been found that mutants can be reduced while preserving testing effectiveness. The state-of-the-art of this problem is the selection of a sufficient set of mutation operators. This can be achieved by omitting the mutation operators that generate most of the mutants [155, 154], or by only including operators that are considered the most effective [207]. A Bayesian selection approach has been recently proposed, that iteratively prioritizes mutation operators with respect to their ability to produce *hard-to-kill* mutants, which make necessary to extend the test suite and thus can improve testing effectiveness [182]. Other approaches randomly select a subset of mutants (*mutation sampling*), or remove mutants that are detected by similar inputs (*mutation clustering*) [106]. Hard-to-kill mutants do not aim to emulate residual faults such as the ones studied in this thesis: while mutants are concerned with the improvement of test suites, SFI aims to emulate faults that escape the software development process in real systems.

Software Fault Injection is used in several (typically post-development) scenarios: to validate the effectiveness and to quantify the coverage of software fault tolerance, to assess risk, to perform dependability evaluation [10, 201, 41, 65]. The application scenario constitutes the first difference from mutation testing and software fault injection: the former is used mainly during software development and is focused on test cases, while the latter is mostly used in post-development scenarios and has a strong requirement of fault representativeness. Since SFI is concerned with the analysis of the system behavior during operation, the conduction of experiments closely emulates the real operational scenario of the target system. Instead of a test set, a workload representative of operational usage is used. Moreover, fault representativeness is a chief concern, in the sense that faults should emulate the residual faults that go with the deployed system.

Compared to mutation operators proposed in the literature for the C language, the fault emulation operators used in Software Fault Injection are more selective and only encompass fault types found in the field: G-SWFIT provides 13 fault types (Table 2.18) against 71 mutation operators proposed in [53]. This reflects the fact that mutation operators inject many kinds of fault that can occur before and during coding and are used to assess the thoroughness of test cases, while fault operators represent faults that escape the whole development process (including testing) and are not designed for improving test suites but assessing fault tolerance. Another difference relies in how fault operators are defined, since they provide additional rules ("constraints") for selecting fault locations in order to better reproduce the fault types observed in the field (see subsection 2.4.3). The proposal of fault operators that reflect the relative occurrence of software faults is instrumental for obtaining a trustworthy evaluation of fault tolerance, and for defining standard and widely agreed procedures for the comparison of software components such as dependability benchmarks [117].

Chapter 3

Improving the representativeness of injected software faults

3.1 Introduction

Fault representativeness is an important concern in Software Fault Injection. As discussed in the previous chapters, the representativeness of faults being injected is needed to obtain confident results from the assessment of fault-tolerant systems, otherwise what is observed from the experiments would not represent what will happen during the operational phase of the system. Moreover, focusing on representative faults (and avoiding to perform non-representative experiments) improves the time and cost effectiveness of the fault injection process.

To achieve fault representativeness, the faultload should emulate the real faults that the system will experience in the field. We refer to these faults as *residual faults*, since they are the ones that escape rigorous design and testing efforts and that actually affect the system during operation. From a practical point of view, fault representativeness implies that

Software Fault Injection should carefully select fault types ("what to inject in the software") and fault locations ("where to inject in the software") to reflect in statistical terms the types and locations of residual faults.

Existing Software Fault Injection approaches [119, 153, 65], including the state-of-the-art technique G-SWFIT, propose a set of realistic fault types, which are derived from the most frequent types of software faults that caused failures of real systems during operation. These approaches neglect the problem of selecting realistic fault locations, and inject faults in every location or randomly select a subset of locations. However, **this aspect is a concern for Software Fault Injection since complex systems have a huge number of locations in which to inject, and only few of them could be suitable to inject a realistic software fault**. It is a matter of fact that residual software faults are not equally likely to exist in every code location, but they are more likely to exist in those modules/routines where the code is more complex and testing activities are less effective [20, 201, 70].

In this chapter, a new Software Fault Injection strategy is proposed for carefully **selecting fault locations to achieve representative faultloads**. The proposed approach is based on the results of an extensive experimental campaign (more than 3.8 million individual experiments) aimed to evaluate and improve the representativeness of injected faults (using the state-of-the-art technique G-SWFIT), as a function of fault types and locations. The definition of representative faultloads is accomplished through the following steps. First, we choose three real world software systems, including two Data Base Management Systems (MySQL and PostgreSQL) and a Real-Time Operating System (RTEMS) largely adopted in business- and safety-critical applications. Second, we conduct extensive SFI campaigns using as workload the *actual test cases adopted by developers*, in order to assess whether injected faults are representative of residual faults or not. The driving idea is that faults disclosed by the test cases do not represent residual faults, as they would be easily detected and fixed by developers. Third, from the results of SFI campaigns we identify the faults that are difficult to find by testing and thus worth considering for SFI, as they are representative of residual faults. Fourth, we conduct a statistical analysis on representative faults to understand how to define a representative faultload for a given software. To this aim, we propose an approach based on classification algorithms and software complexity metrics to identify suitable fault locations for emulating residual software faults. Key findings are:

1. The issue of non-representative faults can significantly affect SFI, even using state-of-the-art techniques, such as G-SWFIT [65]. Considering the experiments done in the RTEMS operating system, it is observed that non-representative faults are the majority of injected faults (72.23%) using the G-SWFIT technique. Even if we consider less-tested and large systems in which the chance of faults to escape testing is higher, such as MySQL and PostgreSQL, the percentage of non-representative faults is still noticeable (respectively, 14.57% and 23.13%).

2. The analysis of the distribution of faults across components (files and modules) reveals that the representativeness of injected faults is significantly affected by fault locations. This result confirms that the careful selection of fault locations is required to improve faultload representativeness.

3. Fault locations can be selected in an effective way by using classification algorithms and software metrics. The proposed approach is a novel compared to existing ones that focus on the selection of fault types [119, 153, 65]. We evaluated both a supervised (i.e., trained using examples) algorithm, namely decision trees, and an unsupervised one, namely k-means clustering. In particular, we found that the faultload can be improved using either the supervised algorithm (4.10%-26.08%) or the unsupervised one (2.16%-16.24%). At the same time, the proposed approach can significantly reduce the faultload size (filtering out up to 69.43% of faults), thus reducing the cost and the time of SFI campaigns in complex software.

This chapter is organized as follows. Section 3.2 presents the experimental evaluation of the representativeness of faults injected by G-SWFIT on three systems. Section 3.3 discusses how representativeness can be improved and shows that there is a clear difference in the distribution of representative/non-representative faults across files and functions. This result leads to the proposal, in Section 3.4, of a new fault selection approach that improves fault representativeness. Section 3.5 summarizes the main contributions of this chapter.

3.2 Evaluation of fault representativeness

This section presents an evaluation of representativeness of the faults injected by G-SWFIT in complex software. In the remainder of this section we discuss the details and the results of this analysis on three case studies. In particular, we analyze the ability of injected faults to escape testing, as they should emulate residual faults that escaped testing and that manifest themselves during the operational phase. The analysis consists of the following steps:

1. We apply G-SWFIT to generate faulty versions of the systems under study. The targets are mature programs that are already well tested and for which real test suites are available.

2. For each injected fault, we evaluate its ability to escape testing (since residual faults, which we aim at emulating, escape testing by their own nature) by running the target with the provided tests cases. Each injected fault will cause a number of the test cases to fail (i.e., the fault is detected). A key aspect here is the fact that we are using the same test cases as the development team of the target system, in order to gain insights about how difficult to detect is a fault.

3. We evaluate if each injected fault can be considered representative or not. If the fault is detected by many test cases, we can assume that the fault is not representative as it is easily discovered by testing. If the fault is not detected by most of the test cases,

then we can assume that the fault is hard to discover and representative of residual faults.

3.2.1 Systems used in the case studies

The case studies considered in this analysis are the MySQL and PostgreSQL DBMSs, and the RTEMS Real-Time Operating System. MySQL is one of the most used DBMSs, accounting for a large share of installations among IT organizations [45]. PostgreSQL is also widely used, including many commercial database applications [68]. RTEMS is an open-source RTOS targeted at embedded systems, and it is also adopted in safety-critical systems [102]. The three software systems considered in our analysis are adopted in real business- and safety-critical contexts, and are a potential target for fault injection (see also past works on fault injection in OSs and DBMSs discussed in Chapter 2 [33, 153, 59, 7, 116, 198]).

Table 3.1: The case studies used in this analysis.

	LoC	**Files**	**Functions**	**Test cases**	**Statement coverage**
MySQL	231,851	223	10,426	469	76.30%
PostgreSQL	366,844	585	9,863	122	66.39%
RTEMS	5,863	555	828	151	96.41%

Software characteristics are depicted in Table 3.1. Statement coverage of test suites was measured using the GCC 3.4.4 compiler and the GCOV tool [164]. MySQL (v. 5.1.34) is made up of more than 230K Lines of Code (LoC) distributed among 223 files and a little over than 10K functions. PostgreSQL (v. 9.0.1) has more than 360K LoC distributed among 585 files and nearly 10K functions. RTEMS (v. 4.9.4) is not as large as the two DBMSs; however, it is still complex software, and, most important, it is supplied with test cases covering more than 96% of the code (running in the QEMU x86 emulator [22]). For the DBMSs, we focus on the DBMS engine, which is the largest and most fundamental part of the DBMS (it is in charge of managing threads and connections, SQL query parsing and optimization); other parts are not considered (e.g., client code, additional plug-ins). Regarding RTEMS, we strictly focus on the kernel code (including task scheduling, time and synchronization, memory management), and do not consider library code (e.g., C library, networking).

All these systems are provided with source code and test cases. Test cases are actually adopted by developers for automating functional and regression testing, and they are augmented as new functionalities are added or unknown faults are found. Test cases are grouped based on the specific part of the system or functionality under test, and we consider only the test cases targeted at the part of the systems we focus on. Since many experiments are conducted for each test case (one experiment per faulty version and test case, see Table 3.2), we selected a sample of 50 test cases for each case study. This sampling reduces the time required for experiments, and can still provide insights about how difficult is to detect faults. Test cases were randomly sampled, and we checked that selected test cases were not too similar. Moreover, test cases achieve at least 50% of statement coverage for all systems. In the case of DBMSs, test cases populate a database and perform several

SQL commands with different variants; they also test specific functionalities of the DBMSs such as triggers and stored procedures. In the RTEMS case study, test cases define a set of tasks to exercise real-time scheduling and system calls. All test cases provided with the case studies are correctly executed (i.e., the system passes the tests in no fault is injected).

Table 3.2: The case studies used in this analysis.

	Faults	**Test cases**	**Statement coverage**	**Total experiments**
MySQL	39,539	50	51.12%	1,976,950
PostgreSQL	32,915	50	57.91%	1,645,750
RTEMS	3,962	50	71.52%	198,100

3.2.2 Experimental Software Fault Injection setup

We used an automated fault injection tool to handle the experiments of this study [148]. The tool injects software faults in a program according to the most common fault types (Table 2.18) found in the field [65]. The tool adopts the same fault operators of G-SWFIT, although faults are introduced in the source code instead of the binary code (Figure 3.1). First, a C pre-processor translates all the C macros in a source code file (e.g., “include” directives), producing a self-contained compilation unit. A C/C++ front-end then analyzes the file and builds an Abstract Syntax Tree representation of the code. This representation guides the identification of locations where a fault type can be introduced in a syntactically correct manner, and that comply to fault type constraints (see Subsection 2.4.3). The tool produces a set of faulty source code files, each containing a different software fault (*faulty* *versions*). Each faulty version is then compiled.

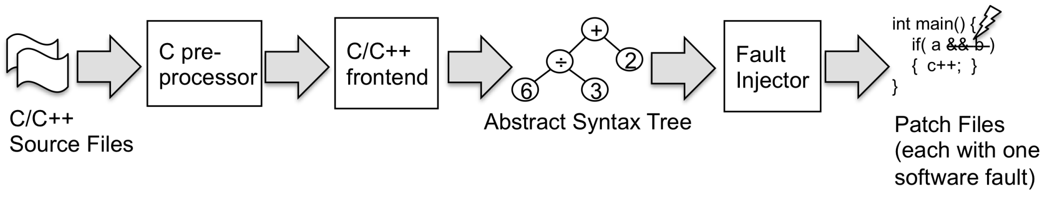


Figure 3.1: Process for generating faulty versions of the target program.

Among the faults generated by the tool, we consider faults in the parts of the system exercised by at least one test case (i.e., source files that are covered during execution). This choice reduces the bias of test case selection, since we draw conclusions about representativeness of faults in the modules that are targeted by the selected test cases. Table 3.2 reports the number of injected faults and experiments for each case study. More than 76 thousands faults were injected, and a total of 3.8 million experiments were performed, which is a very large number when compared to experiments typically found in the literature, and which brings confidence on the validity of results.

The experimental setup is shown in Figure 3.2. In each experiment, the Test Manager executes a test case on a faulty version and collects the test result. Since we are interested in whether the test case is able to detect a given fault (i.e., to cause a failure), we only need a simple failure model (i.e., a pass/fail outcome). DBMS failures are the crash of the DBMS, an incorrect answer to an SQL query, and the timeout of the test. RTEMS failures are the crash of the system or task running, an incorrect output, and the timeout of the test case.

Experiments were performed on 4 workstations equipped with an Intel Core 2 Duo 2.4GHz CPU, 4 Gb RAM, and a SATA 3 Gb/s disk.

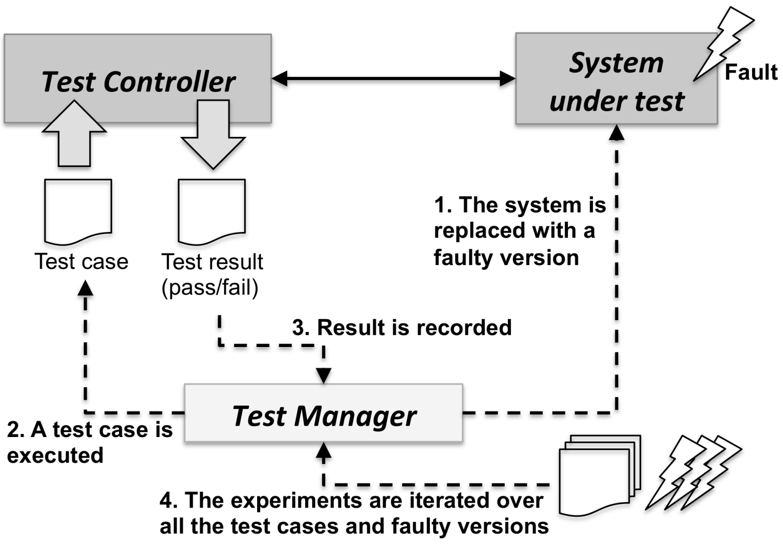

Test Controller
System under test
Fault
Test case
Test result (pass/fail)
1. The system is replaced with a faulty version
3. Result is recorded
Test Manager
2. A test case is executed
4. The experiments are iterated over all the test cases and faulty versions

Figure 3.2: Overview of experimental campaigns.

3.2.3 Result discussion

We analyzed the number of test cases that were able to detect the existence of each fault, in order to identify which injected faults can be considered representative. We also analyze whether the fault location has been executed during a test, by collecting data about statement coverage produced by testing tools. Figure 3.3 shows examples of outcomes occurred in our analysis (not related to a specific system). The horizontal axis represents injected faults (F_1, F_2, and F_3); the vertical axis provides:

1. the percentage of test cases that activated the fault and caused a failure (dark gray);

2. the percentage of test cases that did not detect the fault (i.e., no failure observed), and executed the fault location at least one time (light gray);

3. the percentage of test cases that did not detect the fault and never executed the fault location (white).

Since a faulty version is run against all the 50 test cases selected for that system, it can cause a number of failures from 0 to 50. For instance, from the figure it can be noted that fault F_1 is detected by 1 out of 50 test cases, fault F_2 is detected by 3 out of 50 test cases and its fault location is covered by 40 out of 50 test cases, and fault F_3 is detected by almost all test cases (45 out of 50).

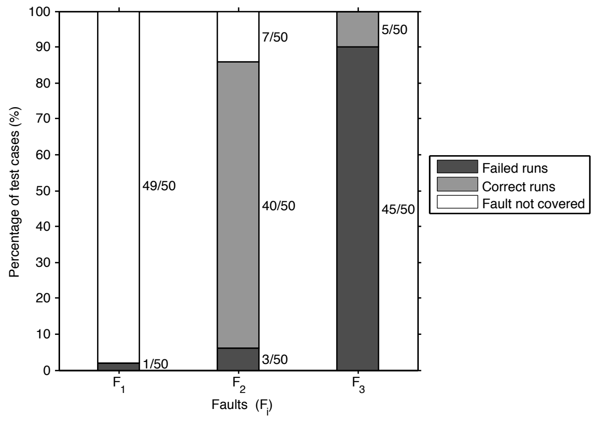


Figure 3.3: Examples of analysis of injected faults with respect to the percentage of failed and correctly executed test cases.

The results are shown in Figure 3.4 (DBMSs) and Figure 3.5 (RTEMS). Faults are ordered by percentage of failures; due to the high number of faults, bars are displayed as lines. A significant part of the faults is detected by most of the test cases (i.e., by more than 50% test cases): 14.57% and 23.13% for DBMSs, and 72.23% for RTEMS (faults on the right side of the axis). These faults should be considered as non-representative; given that the test suites are adopted by developers for detecting faults before a release, we can say that faults that easily cause the system to fail should not be considered as representative. This behavior does not resemble residual faults, which are not caught by testing and remain in the released product.

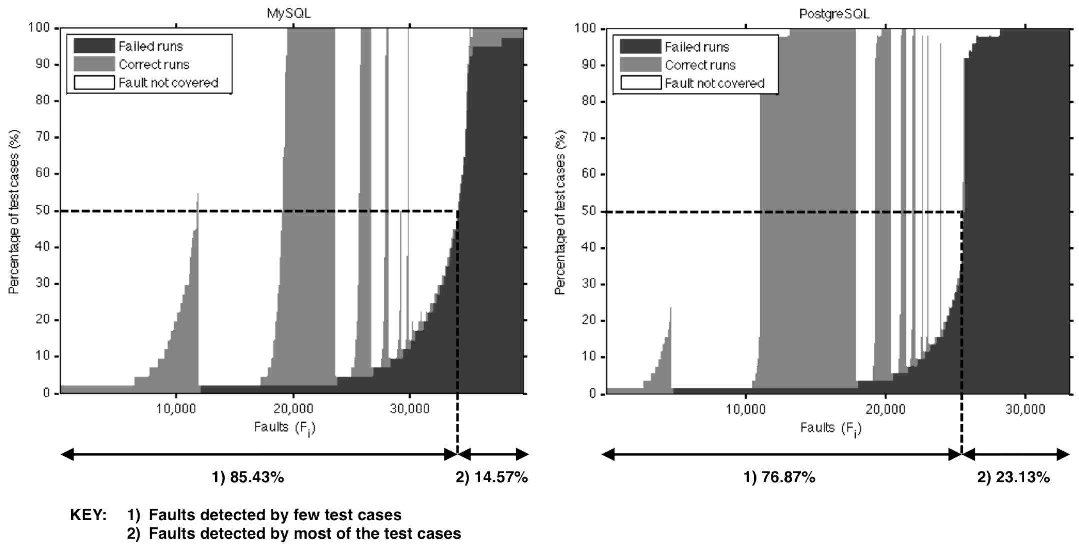


Figure 3.4: Analysis of injected faults in MySQL and PostgreSQL.

Conversely, faults that hardly cause any failure are much more difficult to detect. Part of these faults (the ones under the gray areas) tends to remain undetected even if their location is executed many times. They cause a failure only when the faulty location is executed under specific conditions, which could be easily missed during testing. For instance, the failure

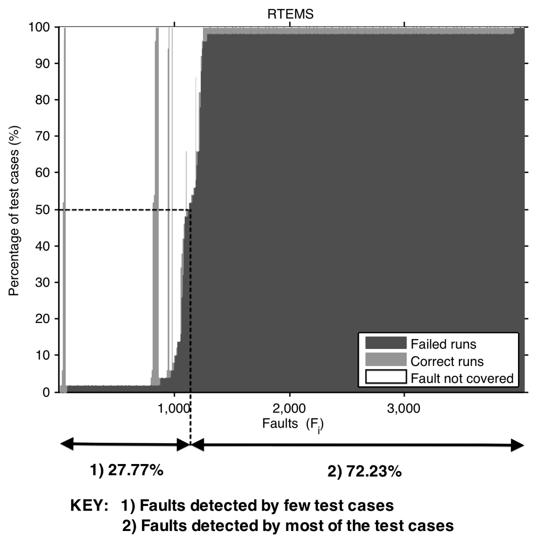


Figure 3.5: Analysis of injected faults in RTEMS.

condition can be related to specific values took by input and state variables. The remaining faults (the ones under the white areas) are detected only by a small number of test cases since the fault location is not executed in the remaining test cases. The portions of code where they reside are hard to cover and exercise, and therefore faults injected there are prone not to be detected by testing activities. These faults should be considered as representative of residual faults existing in the field and useful to inject for assessing fault tolerance.

In order to identify more precisely which faults are "detected by few test cases" and those "detected by many test cases", we analyzed how the percentage of representative faults varies with the threshold value used to discriminate between these two cases. The resulting chart is represented in Figure 3.6. The horizontal axis represents the threshold value; the vertical axis represents the percentage of representative faults detected by a number of test cases below the threshold.

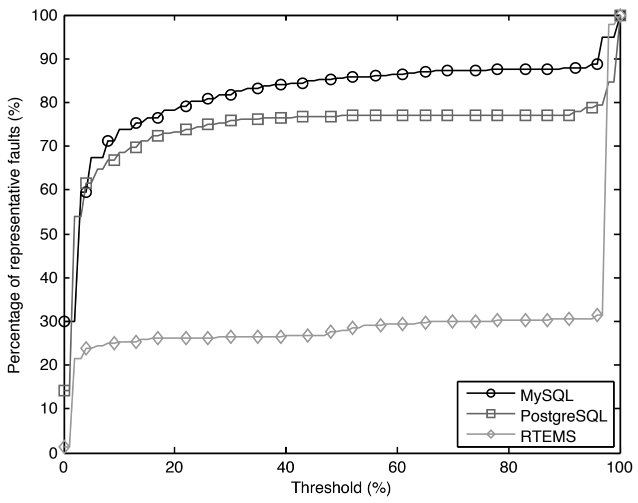


Figure 3.6: Percentage of representative faults across threshold values.

It can be noted that the curves sharply increase when the threshold is below 20%, and then stabilize around a fixed value. The curves sharply increase again when the threshold is over than 90%. This behavior means that in the majority of faults is detected by less than 20% of tests (these faults can be regarded as representative), or by more than 90% of tests. This can be noticed in Figure 3.4 and Figure 3.5. Conversely, only a minority of faults is detected by a percentage of tests between 20% and 90%. Therefore, faults detected by "few" and "many" test cases can be easily identified, that is, the identification of "representative faults" is negligibly affected by the choice of the threshold. Since there is no further evidence that could support the choice of a specific threshold, we opted for the simplest choice of considering half the number of test case sets that we used in our study. Using this threshold, we can see that 85.43% of the injected faults in MySQL are representative as they are detected by less than 50% of the test cases. We can also observe that 76.87% of the faults injected in PostgreSQL are representative, and that only 27.77% of the fault injected in RTEMS are representative. The difference between the DBMSs (which have similar values) and the RTOS is reasonable: MySQL and PostgreSQL are similar systems, and their test coverage is also similar and not as high as the coverage of the test cases of a much smaller system (in code size) such as RTEMS. In fact, RTEMS has a high test coverage, making harder to inject representative faults into it.

3.2.4 Validation of fault representativeness

The results previously presented are based on the assumption that faults escaping the set of test cases are able to represent residual faults that are shipped with the software. However, the faults could still be easily detected before release by using other kind of workloads not necessarily included in the test cases, since test cases tend to assess a specific functionality and not the system as a whole. If this were true, the faults that we consider as representative would be easily detected by using a more complex and comprehensive workload. In order to validate our results, we performed an additional SFI campaign using an implementation of the TPC-C benchmark [48] as a workload for the MySQL case study (we focused only on MySQL due to space and time constraints). The TPC-C performance benchmark is an On-Line Transaction Processing (OLTP) workload that includes a mixture of read-only and update intensive transactions that emulate the activities found in complex OLTP application environments. With TPC-C the DBMS is now being exercised with a long-running and more demanding workload in terms of resources and data manipulation.

We selected one third of the faults that in the previous experiments were detected by at most three test cases. They are the faults that are most difficult to discover (because they seldom cause a failure), therefore we expected that most of them will not be detected in this test. We randomly selected 4 samples of the injected faults, described in Table 3.3. For instance, *Sample 2* includes a third of the faults that caused exactly 2 failed test cases. For each selected faulty version, the TPC-C workload has been continuously executed for 30 minutes. The percentage of failures observed when executing TPC-C is given in the rightmost column. From these results it can observed that faults that were difficult to find using the developer test suite were also difficult to find using a more stressful workload (the TPC-C benchmark). This result supports the assumption that faults avoiding test cases are difficult to find, and our use of test cases to decide if faults are representative.

Table 3.3: Faults and failures using TPC-C.

	Faults in the sample	**% TPC-C Failures**
***Sample 0* (0 failed tests)**	3,960	0.96%
***Sample 1* (1 failed tests)**	3,775	3.31%
***Sample 2* (2 failed tests)**	993	4.03%
***Sample 3* (3 failed tests)**	480	4.58%
All samples	9,280	2.44%

3.3 Improvement of fault representativeness

Results of the previous section gave evidence that SFI campaigns can be affected by a significant amount of faults that are not representative. However, it is not feasible for practitioners to conduct a campaign such as the ones in Section 3.2 to identify which faults are representative for a given system. Therefore, we devise a method to identify representative faults with no need to perform a preliminary experimental analysis. In this way, we would be able to keep fault injection campaigns both feasible and accurate. Since a fault being injected is characterized by its type ("what to inject") and by its location ("where to inject"), we have to assess the relationships between these characteristics and fault representativeness. This would allow to improve the selection of faults to inject, by identifying beforehand which faults are representative and which are not. In the remainder of this section and in the next one we explore how to perform this fault selection. The first step is to understand if and how representative faults can be identified, by analyzing i) the distribution of representative/non-representative faults across fault types, and ii) the representative/non-representative fault distribution across source code locations.

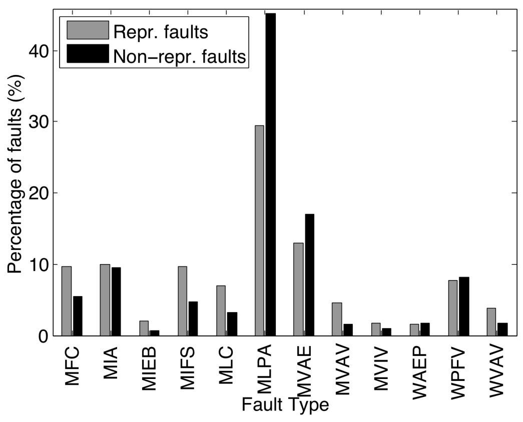
3.3.1 Representativeness across fault types

Figure 3.7 depicts the distribution of representative and non-representative faults across fault types. If fault representativeness were influenced by fault types, we would observe a difference between these distributions. In order to quantitatively evaluate if differences are statistically significant (i.e., they are not caused by random factors), we performed a statistical test, by testing the null hypothesis H_0 that the faults follow the same distribution. To this purpose, we adopted the two-sample Kolmogorov-Smirnov (KS) test [179], which is a non-parametric procedure[1] for evaluating if two samples are drawn from the same underlying probability distribution. The results of the test are reported in Table 3.5; p-values are the probability that observed differences could occur, given that the null hypothesis is true. The test confirms that for all systems the distributions are the same (i.e., H_0 cannot be rejected), with a reasonable degree of confidence (e.g., to reject the null hypothesis with a 90% or 95% significance level, p-values should be lower than 0.1 or 0.05, respectively). Thus there is no statistically significant difference in the distributions of representative/non-representative faults across fault types. Thus, **the results confirm that the type of faults does not affect the representativeness of the faultload**.

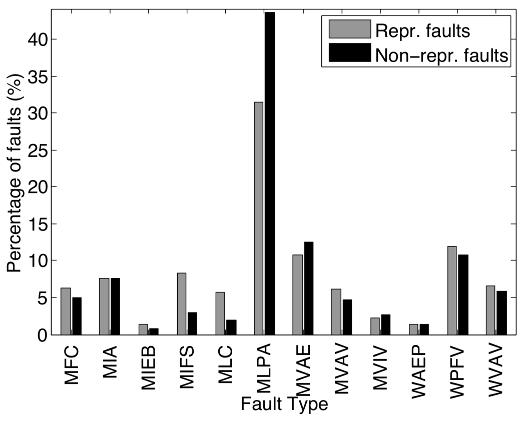
Table 3.4: Kolmogorov-Smirnov likelihood test for representative and non-representative fault distributions across fault types.

Null Hypothesis	**MySQL**	**PostgreSQL**	**RTEMS**
Same distribution across fault types (p-value)	0.4333	0.4889	0.9950
Reject Null Hypothesis	No	No	No

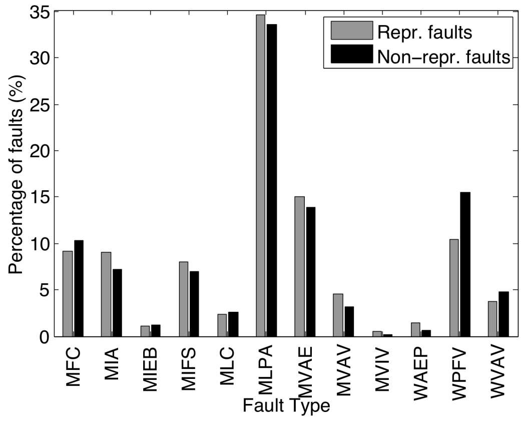
[1]This test was preferred over parametric procedures, such as the t-test, in order not to rely on assumptions about distributions (e.g., normal distributions with same variance).



(a) MySQL.



(b) PostgreSQL.



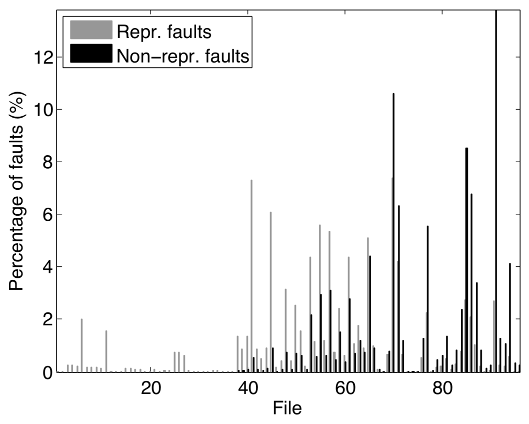
(c) RTEMS.

Figure 3.7: Fault distributions across fault types.

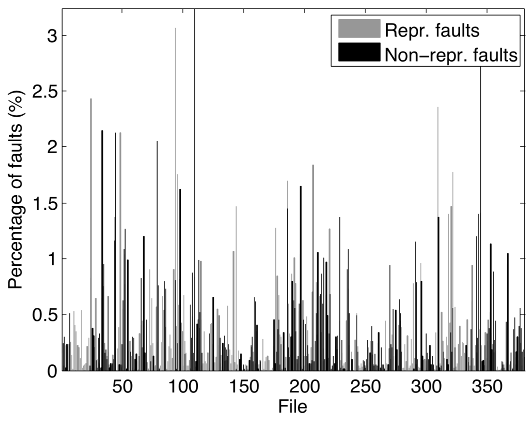
3.3.2 Representativeness across components

As in the case of fault types, we test if there is a statistically significant difference in the distribution of representative/non-representative faults across locations. In particular, we considered fault distributions across source code files and functions of the target systems. Figure 3.8 presents the histograms of fault distributions for each case study. We test the null hypothesis H_0 that the distributions are the same. Table 3.5 presents the resulting p-values.

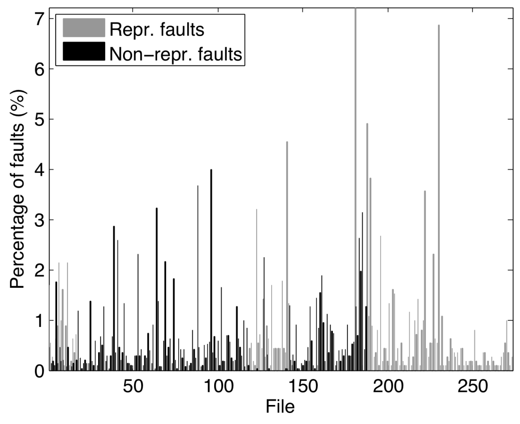
All the p-values obtained are extremely small (less than 0.0001), so we can reject the null hypothesis with a high confidence degree and conclude that **there is a significant difference in the distribution of representative/non-representative faults across components (both files and functions)**. The focus of the next section will be the identification of locations more likely to have representative faults, in order to focus fault injection on them.



(a) MySQL.



(b) PostgreSQL.



(c) RTEMS.

Figure 3.8: Fault distributions across files.

Table 3.5: Kolmogorov-Smirnov likelihood test for representative and non-representative fault distributions across components.

Null Hypothesis	MySQL	PostgreSQL	RTEMS
Same distribution across files (p-value)	7.2862e-07	1.1742e-20	5.1124e-04
Same distribution across functions (p-value)	< smallest float number	< smallest float number	4.0775e-06
Reject Null Hypothesis	Yes	Yes	Yes

3.4 The proposed fault selection approach

We found in the previous section that there is a relationship between fault representativeness and fault locations, and that in some components the percentage of representative faults tends to be higher than the percentage of non-representative faults. This result is due to complexity of the software and its architecture, since fault activation and propagation through the system is affected by the code surrounding the fault. In order to define more representative faultloads and, at the same time, to reduce the cost of fault injection campaigns (in terms of number of injected faults), we propose an approach for identifying components in which to perform the injection campaign, among the set of all components belonging to the target system. The approach analyzes *software complexity metrics* to decide whether a component is appropriate or not for injecting representative faults. It is based on binary classification algorithms, where software metrics (e.g., size and degree of connection of a component) [71] are the classification features. Classification algorithms are useful for making decisions based on complex data (in this case, software metrics), and have also been adopted in other software engineering problems, such as defect predictors [139] or estimation of software development effort [183]. The approach works as follows:

1. Software metrics are collected for every component (files or functions).

2. A classification algorithm is trained with examples (i.e., components for which the percentage of representative faults is known); this step is unnecessary when using an unsupervised classification algorithm (this aspect is discussed in Section 5.4).

3. The classification algorithm is used to identify those components where most of the injected faults are representative, that will be selected for fault injection.

In the following, we first describe how to characterize components, by detailing which components should be selected and which metrics can be analyzed for component selection (Subsection 3.4.1). We then define evaluation criteria to evaluate the effectiveness of the approach, in terms of faultload representativeness and size (Subsection 3.4.2). Finally, we evaluate two classification algorithms for component selection (Subsections 3.4.3 and 3.4.4).

3.4.1 Characterization of software components

In the context of this study, the objects to classify are represented by components. We introduce two classes:

1. Class "Most Representative" (**MR**): components with high percentage of representative faults. These components are thus suitable to be injected.

2. Class "Least Representative" (**LR**): the components with low percentage of representative faults. Injections on these components should be avoided.

There are two possible criteria for dividing components between MR and LR. The first criterion is to assign to the MR class those components where the percentage of representative faults is higher than a fixed threshold; the remaining components are assigned to the LR class. The second criterion is to divide the components such that the MR class includes the components with a percentage of representative faults above the average, and the remaining are assigned to the LR class. Figure 3.9 shows the division according to the latter criterion: it shows the percentage of representative faults in each component (components are sorted by increasing percentage of representative faults), and the vertical line separates the MR class (components “above the average”, on the right) from the LR class (components “below the average”, on the left).

The latter criterion is adopted in this study to assign a class to components rather than using a fixed threshold on the percentage of representative faults, which would lead to an unbalanced division of the components (in the case of MySQL functions, any threshold less than 100% would lead to a very small LR class), and would not take into account that the notion of "high percentage of representative faults" is dependent on the case study (e.g., for RTEMS, any threshold greater than 0% could be considered "high", since about 50% of components have 0% of representative faults).

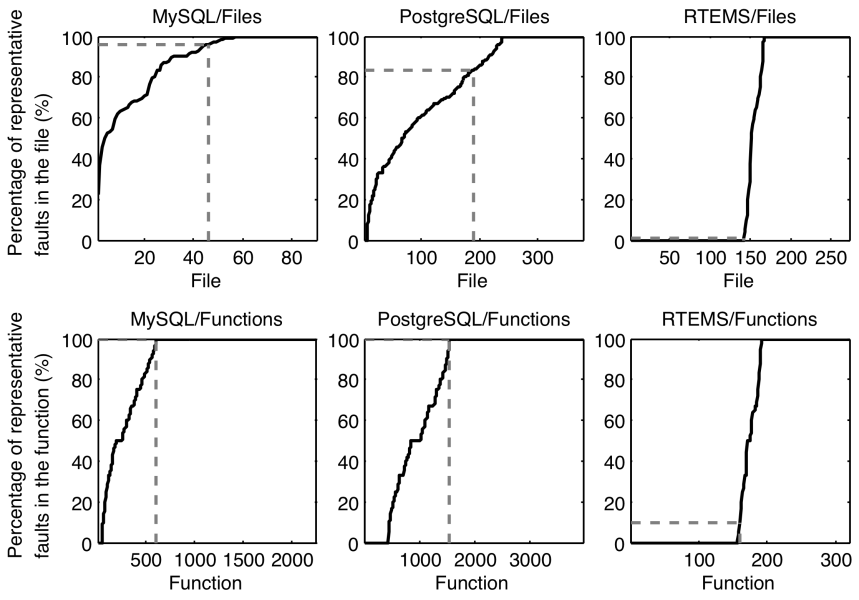


Figure 3.9: Percentage of representative faults for files and functions in the three case studies. The "Most Representative" (MR) components are the points of the X-axis on the right of the vertical line (i.e., percentage of representative faults above the average), and the "Least Representative" (LR) components are those on the left side.

We obtained 6 datasets (two datasets for each case study), which are summarized in Table 3.6. It reports the number of faults in each dataset, and the ratio of representative faults in the set. Columns "All" provide these data for all components in the dataset; the remaining columns are obtained by only looking at components of the MR or the LR class, respectively. Since the MR class is made up of components that have a percentage of representative faults above the average, this class has a higher ratio of representative faults than the full dataset (e.g., in the case of MySQL/files, 98.51% of faults in MR components is representative, against 85.49% when all components are considered). MR percentages represent an upper bound to the improvement that can be gained by perfect component selection, i.e., if all MR components could be correctly identified and the others are discarded. Additionally, MR components represent a subset of the faults, therefore component selection can also lead to smaller faultloads. The approach classifies components of a target system as either MR or LR (of course, the membership of components is unknown before a fault injection campaign).

Table 3.6: Characterization of the datasets (All components, MR components, and LR components).

Dataset	**% of representative faults**			**Number of faults**		
	All	**MR**	**LR**	**All**	**MR**	**LR**
MySQL/Files	85.49%	98.51%	80.65%	39,539	10,708	28,831
MySQL/Functions	85.49%	100.00%	65.62%	39,539	22,816	16,723
RTEMS/Files	28.24%	72.10%	0.00%	3,962	1,158	2,804
RTEMS/Functions	28.24%	82.21%	0.19%	3,962	1,166	2,796
PostgreSQL/Files	77.08%	95.12%	62.04%	32,915	14,969	17,946
PostgreSQL/Functions	77.08%	100.00%	51.75%	32,915	17,248	15,667

A set of metrics (Table 3.7) has been selected for analyzing software complexity, which are commonly used by researchers and practitioners. Lines of Codes and Cyclomatic Complexity represent the number of statements and the number of paths in a component: they are traditionally regarded as indicators of complexity since they characterize the size and structure of functionalities implemented by a program [20, 71, 70]. FanIn and FanOut, which count the connections between components, provide insights the complexity of the system structure and of the information flow among components [71, 92]. We do not consider other metrics such as Software Science (Halstead) and Object-Oriented metrics (e.g., Chidamber-Kemerer), since 1) metrics tend to be correlated with each other, therefore limiting the benefits of considering many metrics [71] (although our approach does not prevent the inclusion of more metrics), 2) some of them are not generic (e.g., they only apply to object-oriented systems), and 3) they cannot be estimated in the absence of source code[2], which is often the case of third-party software. Metrics were collected using the Understand tool [176].

Table 3.7: Software complexity metrics.

Metric	**Description**
Lines of Code (LoC)	The number of executable lines of code in a program. For files, we consider both the average and the total LoC of functions in the file. For functions, we consider the number of lines of code of individual functions.
McCabe's cyclomatic complexity	The number of linearly independent paths through a function. For files, we consider the sum, the average and the maximum cyclomatic complexity of functions in the file.
FanIn and FanOut	The count of unique functions that call (or are called by) a given function, either directly, or ultimately, via other functions. For files, these metrics are based on the unique functions that call (or are called by) any of the functions defined in the file, and exclude calls between functions within the same file.

3.4.2 Evaluation measures

We introduce a set of measures for assessing the ability of the proposed approach to correctly classify components and, ultimately, to improve faultload representativeness. Since the purpose of our approach is to avoid injecting non-representative faults, the primary measure

[2]The binary code can potentially be used for estimating the size of functions and the dependencies between functions, but it lacks information about symbols (e.g., variables) in the source code, which is needed for computing Halstead metrics. We do not focus on how to estimate complexity metrics from binary code, since this aspect is outside the scope of the present work.

for assessing the approach is the percentage of representative faults within the faultload:

$$\%\text{Representative} = \frac{\#\ \text{Representative faults in the faultload}}{\#\ \text{Faults in the faultload}}; \tag{3.1}$$

in particular, we denote with $\%\text{Representative}_{\text{filtering}}$ the percentage of representative faults in the faultload using the proposed approach, and with $\%\text{Representative}_{\text{MR}}$ and $\%\text{Representative}_{\text{LR}}$ the percentage computed for the MR and LR classes, respectively (see Table 3.6). In a similar way, we evaluate the number of faults in the faultload, namely $\#\text{Faults}_{\text{filtering}}$, $\#\text{Faults}_{\text{MR}}$, and $\#\text{Faults}_{\text{LR}}$.

Additionally, we consider measures specifically aimed at evaluating classification algorithms, namely *precision* and *recall* [206]. These measures compare the set of objects that should be selected (i.e., the MR class) with the set of objects actually selected by the classifier (see Figure 3.10). The measures are based on the following quantities, that is, the number of objects correctly or wrongly classified:

1. **True Positives** (TP): Number of MR components correctly identified as MR.
2. **False Positives** (FP): Number of LR components wrongly classified as MR.
3. **False Negatives** (FN): Number of MR components wrongly classified as LR.

In turn, precision and recall are computed as follows:

1. **Precision** $= TP/((TP+FP))$: Percentage of True Positives with respect to the whole set of selected components (which includes both TPs and FPs).

2. **Recall** $= TP/((TP+FN))$: Percentage of True Positives with respect to the set of MR components (which includes both TPs and FNs).

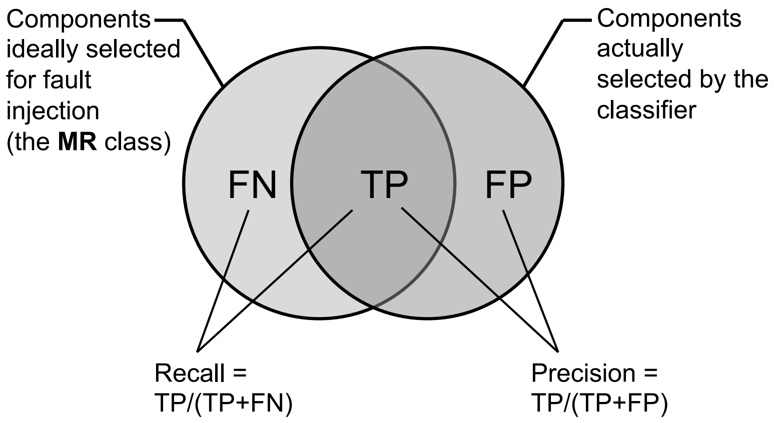

Components ideally selected for fault injection (the **MR** class)
Components actually selected by the classifier
FN
TP
FP
Recall = TP/(TP+FN)
Precision = TP/(TP+FP)

Figure 3.10: Measures adopted for assessing classification algorithms.

Ideally, a classifier should have high precision (as close as possible to 1), to keep low the number of non-representative faults in the faultload (due to False Positives), and they should also have high recall (as close as possible to 1), to keep low the number of representative faults missed (due to False Negatives). Both precision and recall are related to the percentage of representative faults that can be obtained by filtering components through a classifier. The expected percentage of representative faults after selection is

$$\%\text{Representative}_{\text{filtering}} = \text{Precision}\cdot\%\text{Representative}_{\text{MR}} + (1-\text{Precision})\cdot\%\text{Representative}_{\text{LR}}, \tag{3.2}$$

which is the weighted average between the densities of representative faults in MR and LR classes: the higher the precision, the closer the average to the MR class. The expected number of faultload size after filtering is

$$\#\text{Faults}_{\text{filtering}} = \#\text{Faults}_{\text{MR}} \cdot \frac{\text{Recall}}{\text{Precision}}. \tag{3.3}$$

It should be noted that $\#\text{Faults}_{\text{filtering}}$ is inflated with False Positives when Precision is low. When Recall is low, $\#\text{Faults}_{\text{filtering}}$ is low due to False Negatives (i.e., some MR components are not recognized by the classifier).

3.4.3 Fault Selection by using decision trees

The first algorithm that we adopt for component classification is a technique commonly used in data mining problems, namely *decision trees* [206]. A decision tree is a hierarchical set of questions that are used to classify an element. In our study, questions are based on software metrics (for instance "Is LoC greater than 340?"), and the components are the elements to be classified. This algorithm is a supervised classifier, since it requires to be trained with examples in order to classify unknown elements. Decision trees have been preferred over other supervised classifiers because they are simple to interpret, therefore they can provide insights on the relationship between complexity metrics and component classes. This classifier is provided by most machine learning tools, such as the WEKA tool used in this work [206].

A decision tree is obtained from a training dataset using the C4.5 algorithm [206]. The C4.5 algorithm iteratively splits the dataset in two parts, by choosing the individual attribute (i.e., complexity metric) and a threshold that best separates the training data into the classes; this operation is then repeated on the subsets, until the classification error (estimated on the training set) cannot be reduced anymore. The root and inner nodes represent questions about complexity metrics, and leafs represent class labels. To classify a component, a metric of the component is first compared to the threshold specified in the root node, to choose one of the two children nodes; this operation is repeated for each selected node, until a leaf is reached.

The performance of decision trees was evaluated on the 6 datasets (Subsection 3.4.1) through *cross-validation*. Each dataset is divided in a training set (one third of the data) and a test set (two thirds of the data); evaluation metrics (Subsection 3.4.2) are then computed by classifying the components of the test set. Since the dataset split can affect the performance of the classifier, we considered 10 random splits for each dataset. The results of cross-validation are provided in Table 3.8. It is worth noting that the average precision is higher than 0.6 for every dataset (i.e., TPs are more than FPs), therefore we expect that the percentage of representative faults is increased in the filtered faultload. Moreover, since the recall ranges between 0.63 and 0.93, the filtered faultload includes most of the MR components (i.e., TPs are more than FNs) and therefore most of the representative faults. It should be observed that decision trees provide better performance on the "functions" datasets than on the "files" datasets (with respect to all the metrics and the case studies), since a smaller granularity can enable a more precise filtering (e.g., if most of the non-representative faults in a file are in a few functions, only these functions can be removed).

Table 3.8: Performance of decision trees in terms of precision and recall (mean ± standard deviation).

Type of component	**MySQL**		**PostgreSQL**		**RTEMS**		**Average**	
	Precision	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
Files	0.63 ± 0.09	0.74 ± 0.14	0.65 ± 0.02	0.68 ± 0.14	0.61 ± 0.11	0.63 ± 0.24	0.63 ± 0.08	0.68 ± 0.18
Functions	0.77 ± 0.03	0.93 ± 0.04	0.66 ± 0.02	0.87 ± 0.08	0.61 ± 0.07	0.64 ± 0.19	0.68 ± 0.08	0.81 ± 0.17

Table 3.9 analyzes the faultload size and percentage of representative faults when using decision trees. The first two columns describe the original datasets (they report data from Table 3.6 for the sake of readability). The last two columns describe the expected number of faults and ratio of representative faults when a subset of components is selected using a decision tree with precision/recall as estimated through cross-validation (see Table 3.8 and Eq. 3.2 and 3.3). Faultload representativeness increases in all cases (see the column on the right side). The improvement is more significant for RTEMS (up to 26.08%), since the difference between $\%\text{Representative}_{\text{MR}}$ and $\%\text{Representative}_{\text{LR}}$ (Table 3.6) is more

significant in this case study (e.g., there are several components with 0% of representative faults), therefore the benefit of faultload filtering is greater in this case. After filtering, a large share of faults is removed from the faultload (between 30.30% and 69.43%); at the same time, due to the high recall, we are confident that most of the representative faults in the initial faultload are still present in the filtered faultload.

Table 3.9: Percentage of representative faults before and after component selection using decision trees.

Dataset	**G-SWFIT**		**G-SWFIT + decision trees**	
	Faultload size	**% of repr. faults**	**Faultload size**	**% of repr. faults**
MySQL/Files	39,539	85.49%	12,578 (**-67.94%**)	91.90% (+**6.41%**)
MySQL/Functions	39,539	85.49%	27,557 (**-30.30%**)	92.09% (+**6.60%**)
RTEMS/Files	3,962	28.24%	1,211 (**-69.43%**)	46.87% (+**18.63%**)
RTEMS/Functions	3,962	28.24%	1,537 (**-61.21%**)	54.32% (+**26.08%**)
PostgreSQL/Files	32,915	77.08%	15,460 (**-53.03%**)	82.22% (+**5.14%**)
PostgreSQL/Functions	32,915	77.08%	18,096 (**-45.02%**)	81.18% (+**4.10%**)

Figure 3.11 shows the decision trees that are automatically learned using the C4.5 algorithm from the "function" datasets. Leafs contain the class labels (MR and LR), along with the number of components of the dataset that are correctly and wrongly classified by the leaf, respectively. By analyzing the structure of the tree, it can be noticed that the complexity metrics involved in the classification are FanIn, FanOut, and LinesOfCode, and that the cyclomatic complexity is not present. This is due to a correlation existing between cyclomatic complexity and the other metrics, which makes it a redundant metric [71]; it can also be an artifact of this particular classification algorithm. The FanIn and FanOut seem

to be the metrics most relevant for discriminating between components:

1. In the MySQL case study, most of the MR components (1,189 out of 1,653) have FanIn lower than 62; conversely, several LR components (401 out of 604) have FanIn greater than 62.
2. In the PostgreSQL case study, most of the MR components (2,041 out of 2,418) have FanIn lower than 962; conversely, several LR components (436 out of 1,539) have FanIn greater than 962.
3. In the RTEMS case study, most of the MR components (110 out of 159) have FanOut lower than 15; conversely, several LR components (98 out of 161) have FanOut greater than 15.

The relevance of FanIn and FanOut might be explained by the higher "exposure" of faults in a component with a large number of connections to other components; it is thus more difficult to inject “difficult-to-detect” faults in these components, and faultload representativeness can be improved by looking at these metrics.

3.4.4 Fault selection by using clustering

In the previous subsection, we concluded that complexity metrics can be exploited to identify components in which to inject faults, using a supervised classifier. However, the main limitation of that algorithm is represented by the need for a training set, since getting a training

```
FanIn <= 62: MR (1,189 correct, 203 wrong)
FanIn > 62
|  LinesOfCode <= 81
|  |  FanIn <= 195: MR (300 correct, 160 wrong)
|  |  FanIn > 195: LR (171 correct, 142 wrong)
```

(a) MySQL.

```
FanIn <= 962
|  FanIn <= 93
|  |  FanIn <= 22: MR (737 correct, 280 wrong)
|  |  FanIn > 22
|  |  |  FanIn <= 36: LR (93 correct, 42 wrong)
|  |  |  FanIn > 36
|  |  |  |  LinesOfCode <= 21
|  |  |  |  |  FanIn <= 74: MR (40 correct, 8 wrong)
|  |  |  |  |  FanIn > 74: LR (3 correct)
|  |  |  |  LinesOfCode > 21: LR (37 correct, 26 wrong)
|  FanIn > 93: MR (1,294 correct, 682 wrong)
FanIn > 962: LR (436 correct, 279 wrong)
```

(b) PostgreSQL.

```
FanOut <= 15
|   FanIn <= 4
|   |   FanIn <= 0: MR (5 correct, 1 wrong)
|   |   FanIn > 0: LR (15 correct, 2 wrong)
|   FanIn > 4: MR (103 correct, 47 wrong)
FanOut > 15
|   FanIn <= 813: LR (94 correct, 38 wrong)
|   FanIn > 813
|   |   FanOut <= 61: MR (11 correct, 1 wrong)
|   |   FanOut > 61: LR (3 correct)
```

(c) RTEMS.

Figure 3.11: Decision trees learned from the "function" datasets.

set would require an experimental analysis similar to the one in Section 3.2. Therefore, a practitioner would need to identify representative faults for a subset of components, to train the algorithm, and to classify the remaining components. To overcome this limitation, we consider an unsupervised classifier (i.e., that does not require a preliminary training phase), namely a *clustering algorithm*. This approach is based on the finding that MR components have low FanIn or FanOut and tend to be aggregated below a threshold.

A clustering algorithm partitions a dataset into subsets (*clusters*) such that data in each subset are similar (according to some *distance measure*). Therefore, it can be used to partition the components into two sets, and then to select only one subset for fault injection. Two aspects need to be defined to adopt this strategy: a distance measure, and a criterion for selecting the target cluster.

- **Define a distance measure**: we define the distance measure as the euclidean distance in the space of software complexity metrics, in order to discriminate between "low" and "high" values of software metrics. In the following, we evaluate several combinations of software complexity metrics for this purpose; we focus on LinesOfCode, FanIn, and FanOut, since they turned out to be the most relevant metrics in the previous analysis.
- **Define a criterion for selecting the target cluster**: a clustering algorithm can split the data set in two subsets; however, only one subset has to be selected for fault injection. Since the MR components are characterized by the lowest FanIn and FanOut, we select the cluster in which to inject faults by computing the mean value of FanIn and FanOut of components in each cluster. We then select the cluster with the lowest average FanOut or the lowest average FanIn (both criteria were evaluated).

Among the clustering algorithms proposed in the literature, we adopt the *Lloyd k-means clustering algorithm*, which is well known and simple to understand [206]. K-means clustering identifies k clusters that minimize the variance of distance of elements within the same cluster. In our approach, we adopt the fixed value $k = 2$ when applying clustering (even if the samples could be divided in more clusters), since we aim at discriminating between only two classes. The algorithm is an iterative procedure. It randomly selects k elements (namely *centroids*), each representing the “mean” of a cluster, and assigns the remaining elements to the cluster of the nearest centroid. The procedure is repeated by computing the means of the clusters obtained in the previous iteration, that are used as new centroids. It stops when clusters do not change between iterations or after a maximum number of iterations. The clustering algorithm is executed 10 times, by varying the random selection of the initial k elements. Data were normalized in the range $[0, 1]$ before clustering.

Table 3.10 and Table 3.11 show the performance of the k-means clustering algorithm with respect to the three case studies; for the sake of brevity, results are only provided with respect to "function" components. For each case study, we evaluated the effectiveness of different distance measures and cluster selection criteria. The best results (in terms of high average and low standard deviation of both precision and recall) are obtained when (i) selecting the cluster with the lowest FanOut, and (ii) the distance measure is based on LoC or FanOut (highlighted in Table 3.10). It can be observed that the clustering algorithm is close to decision trees in terms of precision (respectively, 0.63 and 0.68 in average) and recall (0.81 for both decision trees and clustering using FanOut and LinesOfCode). Instead, cluster selection using the lowest FanIn is not effective in the case of RTEMS (precision is less than 50%), although this criterion still gives good results for MySQL and PostgreSQL. Therefore, although the FanIn was selected by decision trees learned from the MySQL and PostgreSQL datasets, the FanOut turned out to be more useful than FanIn to provide a common selection criteria among all the systems. This result is due to a correlation existing between FanIn and FanOut (i.e., high FanIn and high FanOut tend to occur at the same time, and vice versa). For this reason, both FanIn and FanOut are effective for the MySQL and PostgreSQL systems, but the FanOut metric should be preferred as a generic criterion for identifying components in which to inject faults.

Table 3.10: Performance of clustering (mean ± standard deviation), using FanOut for selecting the target cluster.

Metrics	**MySQL**		**PostgreSQL**		**RTEMS**		**Average**	
	Precision	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
LoC, FanIn, FanOut	0.68 ± 0.12	0.77 ± 0.35	0.56 ± 0.10	0.52 ± 0.27	0.71 ± 0.00	0.28 ± 0.00	0.65 ± 0.11	0.52 ± 0.32
FanIn, FanOut	0.68 ± 0.12	0.76 ± 0.36	0.57 ± 0.08	0.55 ± 0.23	0.71 ± 0.00	0.28 ± 0.00	0.65 ± 0.10	0.53 ± 0.31
LoC, FanOut	0.74 ± 0.00	0.94 ± 0.00	0.61 ± 0.00	0.63 ± 0.04	0.54 ± 0.00	0.86 ± 0.00	0.63 ± 0.08	0.81 ± 0.13
FanOut	0.74 ± 0.00	0.94 ± 0.00	0.61 ± 0.00	0.66 ± 0.08	0.54 ± 0.00	0.86 ± 0.01	0.63 ± 0.08	0.82 ± 0.13
LoC, FanIn	0.46 ± 0.07	0.13 ± 0.20	0.47 ± 0.03	0.23 ± 0.06	0.71 ± 0.00	0.28 ± 0.00	0.54 ± 0.13	0.21 ± 0.14
FanIn	0.44 ± 0.00	0.08 ± 0.00	0.46 ± 0.04	0.22 ± 0.07	0.71 ± 0.00	0.28 ± 0.00	0.54 ± 0.13	0.19 ± 0.09
LoC	0.74 ± 0.00	0.99 ± 0.00	0.62 ± 0.00	0.95 ± 0.00	0.51 ± 0.00	0.87 ± 0.01	0.63 ± 0.09	0.94 ± 0.05
Average							0.61 ± 0.11	0.57 ± 0.34

Figure 3.12 shows the distribution of components with respect to FanOut and LinesOfCode metrics. The cross marks represent the centroids of the clusters; a cluster is composed

Table 3.11: Performance of clustering (mean ± standard deviation), using FanIn for selecting the target cluster.

Metrics	**MySQL**		**PostgreSQL**		**RTEMS**		**Average**	
	Precision	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
LoC, FanIn, FanOut	0.70 ± 0.06	0.36 ± 0.42	0.62 ± 0.01	0.45 ± 0.23	0.44 ± 0.00	0.72 ± 0.00	0.59 ± 0.11	0.51 ± 0.31
FanIn, FanOut	0.68 ± 0.05	0.23 ± 0.35	0.63 ± 0.01	0.46 ± 0.22	0.44 ± 0.00	0.72 ± 0.00	0.58 ± 0.11	0.47 ± 0.31
LoC, FanOut	0.65 ± 0.00	0.06 ± 0.00	0.62 ± 0.00	0.35 ± 0.07	0.35 ± 0.00	0.14 ± 0.00	0.54 ± 0.14	0.18 ± 0.13
FanOut	0.65 ± 0.00	0.06 ± 0.00	0.62 ± 0.00	0.34 ± 0.08	0.33 ± 0.00	0.14 ± 0.01	0.54 ± 0.14	0.18 ± 0.13
LoC, FanIn	0.78 ± 0.00	0.92 ± 0.00	0.66 ± 0.01	0.78 ± 0.07	0.44 ± 0.00	0.72 ± 0.00	0.63 ± 0.14	0.81 ± 0.09
FanIn	0.78 ± 0.00	0.92 ± 0.00	0.66 ± 0.02	0.80 ± 0.09	0.44 ± 0.00	0.72 ± 0.00	0.62 ± 0.14	0.81 ± 0.10
LoC	0.74 ± 0.00	0.99 ± 0.00	0.44 ± 0.01	0.05 ± 0.00	0.51 ± 0.00	0.87 ± 0.01	0.57 ± 0.13	0.64 ± 0.43
Average							0.58 ± 0.13	0.51 ± 0.34

by the nearest elements to its centroid. The cluster to be selected is the one on the bottom-left corner of the plots (lowest FanOut). The high precision and recall of the clustering algorithm is due to the density of MR functions being higher than the density of LR functions in that cluster; in other words, when only the target cluster is selected, a high amount of MR functions is retained, while several LR functions are avoided at the same time.

Table 3.12 summarizes the results of the clustering algorithm in terms of percentage of representative faults and faultload size. The clustering algorithm is able to improve the faultload representativeness, and the best results are achieved when clustering at the "function" granularity (the improvement ranges between 4.10% and 16.24%). Compared to decision trees, the clustering algorithm provides a lower performance; nevertheless, the clustering algorithm does not require to be trained using examples (which can be costly to obtain), since it exploits the relationship between the FanOut metric and fault representativeness

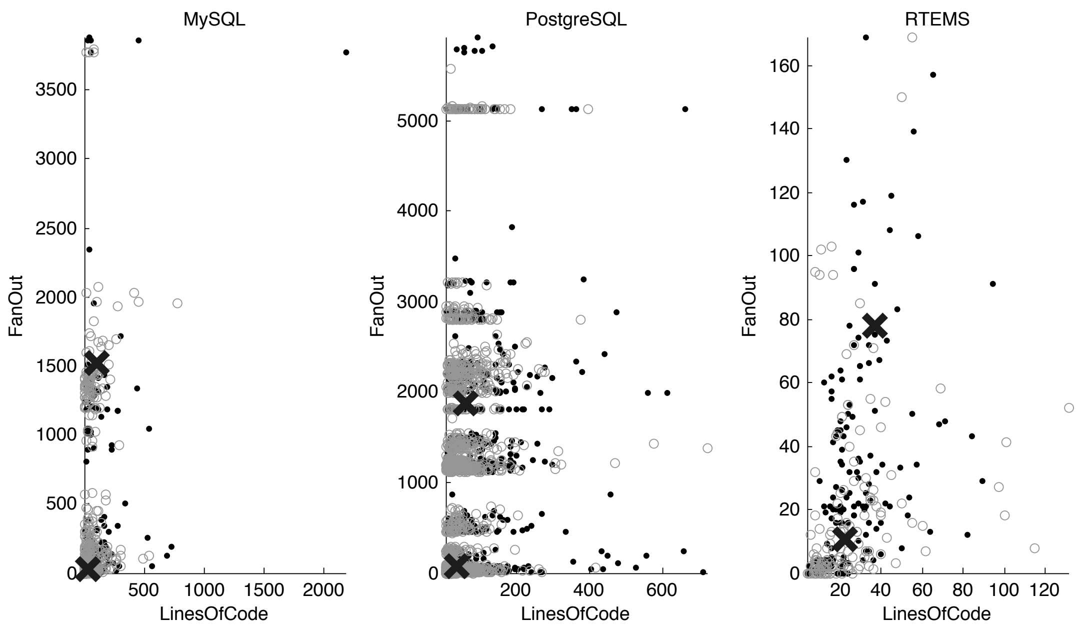


Figure 3.12: Scatter plot of MR (non-filled circles) and LR (filled circles) functions with respect to LinesOfCode and FanOut. Cross marks identify cluster centroids.

(shown in Figure 3.11). Therefore, clustering is a valuable approach for both improving representativeness and keeping experimental effort low.

Table 3.12: Percentage of representative faults before and after component selection using clustering.

Dataset	**G-SWFIT**		**G-SWFIT + clustering**	
	Faultload size	**% of repr. faults**	**Faultload size**	**% of repr. faults**
MySQL/Files	39,539	85.49%	16,159 (**-59.13%**)	90.47% (+**4.98%**)
MySQL/Functions	39,539	85.49%	28,982 (**-26.70%**)	91.06% (+**5.57%**)
RTEMS/Files	3,962	28.24%	1,816 (**-54.16%**)	36.77% (+**8.53%**)
RTEMS/Functions	3,962	28.24%	1,857 (**-53.13%**)	44.48% (+**16.24%**)
PostgreSQL/Files	32,915	77.08%	25,620 (**-22.16%**)	79.24% (+**2.16%**)
PostgreSQL/Functions	32,915	77.08%	17,813 (**-45.88%**)	81.18% (+**4.10%**)

3.5 Summary

In this chapter, we analyzed the representativeness of injected faults in three complex, real-world software systems, and proposed an approach for improving fault representativeness. This aspect is important for obtaining a realistic assessment of fault tolerance. The SFI technique considered in this thesis, G-SWFIT, aims to achieve fault representativeness by emulating the most frequent fault types found in operational systems. In this chapter, we analyzed fault representativeness with respect to an additional criterion, that is, the ability of faults to escape *real test suites*, which characterizes residual faults that affect operational systems and that should be tolerated.

After analyzing a large set of injected faults and real test cases (up to 3.8 million experiments), we concluded that the percentage of representative faults ranges from a minor share in the case of DBMSs (14.57% and 23.13%) to a significant share in the case of a RTOS (72.23%). The proposed approach selects a subset of components suitable for injecting representative faults, by analyzing their complexity and relationships using classification algorithms and software metrics. The first considered algorithm, *decision trees*, is a supervised classifier, which is trained by providing examples of components to be selected. The second algorithm, *k-means clustering*, is an unsupervised classifier, which does not require to be trained with examples but relies on the observation that suitable components have the lowest FanIn and FanOut, as they are less exposed to testing. We found that both these algorithms can accurately classify components for all the case studies (ranging from small and well-tested to large and less-tested software), and that it is possible to improve fault representativeness and reduce faultload size at the same time. In the light of these results, the proposed approach can be regarded as an effective and practical means for improving the realism of SFI.

The approach and the results reported in this study are based on empirical data, which limits the generality of the conclusions. The considered case studies are open-source systems, and results could not apply to software developed under other paradigms. Nevertheless, all the considered systems were developed with the support of commercial organizations, and they were tested using common practices also adopted in commercial software. In the cases of MySQL and PostgreSQL, a request- and bug-tracking system is adopted, which is also used to some degree for introducing test cases to cope with new features or documented faults that should be avoided in future development. In the case of RTEMS, the system has been engineered and tested to fulfill the requirements of industrial safety standards, for its adoption in space applications by the European Space Agency [102]. Since there are similarities between these systems and industrial ones, we believe that the results can potentially be extended to other kinds of system.

Chapter 4

Improving the accuracy of software faults injected in binary code

4.1 Introduction

The market competition and time-to-market deadlines impose strict constraints on the time and resources available for software development. For this reason, we are witnessing the massive adoption of *Commercial Off-The-Shelf* software, i.e., third-party software components not developed for a specific system, that software firms buy from others in order to avoid the cost of developing the same functionality by their own. A similar trend towards *code reuse* from previous product versions can be noticed. Industries involved in the development of business- and safety-critical systems are not an exception to these trends. Unfortunately, COTS and reused software are harmful to software dependability, since they are used in a new environment that was not foreseen when they were originally developed, leading to incorrect interactions between the reused component and the environment, or to the activation of hidden faults that never manifested themselves in the past. Moreover, it is unpractical, and often unfeasible, to test and debug them due to the lack of source code and/or expertise on their internals and specification [204].

The widespread use of COTS and the risk of software faults emphasize the importance of Software Fault Injection in this kind of software components. SFI can be adopted to assess during development what will be the impact of software faults in COTS components, and to take countermeasures by adopting fault tolerance mechanisms. However, their source code is often not available and only their *executable code* (also referred to as *binary* or *machine code*) can be accessed. Therefore, it is necessary to be able to inject software faults within the binary code, as in the case of G-SWFIT [65]. This approach requires that programming constructs used in the source code are identified by looking only at the binary code, since the injection is performed at this level.

Unfortunately, binary-level SFI is a difficult and error-prone task due to the complexity of programming languages and of modern compilers, which make difficult and in some cases impossible to accurately recognize where to inject faults. Therefore, an important issue concerning the injection of software faults at binary level is to assure the *accuracy* of the injection campaign, that is, the degree of confidence that a fault injected in the binary code correctly emulates a software defect in the source code. For instance, if we aim to emulate the absence of a variable assignment in the source code, we could remove a "move" instruction at binary level. But, if we consider the emulation of a bug in a C preprocessor macro (i.e., a piece of source code that is replicated several times in the binary code), the problem cannot be resolved by simply looking at the binary code (which lacks information about preprocessor macro).

Assuring the accuracy of binary-level SFI techniques is therefore a major concern for their adoption in real-world scenarios. This assessment is important to gain confidence that results from binary-level fault injection are accurate, and to test, debug and improve SFI techniques and tools with respect to real-world software. Unfortunately, only a few studies evaluated the accuracy of binary-level SFI, which were limited to small programs or to a small number of faults [60, 65, 107], and there is not an approach for analyzing this problem comprehensively.

In this chapter, we propose a method for **assessing the accuracy of binary-level fault injection in complex software**, and perform an extensive experimental campaign in order to **assess the accuracy of G-SWFIT**. The proposed method performs two fault injection campaigns on the same target system, respectively injecting faults in its *binary code* and *source code*, where the latter is used as a term of comparison. During these campaigns we keep track of code locations targeted by fault injection. We then compare for each fault type the locations affected by source-level injection with the ones affected by binary-level injection. In this way, we are able to identify: (i) *binary-level faults which correctly emulate software faults* (this happens when we experience the same fault type in the same location both from binary-level injection and from source-level injection); (ii) *binary-level faults that do not emulate any software fault* (this happens when a binary-level fault is injected in a location in which the fault could not exist in the source code); and (iii) *binary-level faults that have not been injected in a location where they could have been injected.*

The experimental evaluation of G-SWFIT consists of the injection of about 30 thousand faults, 12 thousand binary-level faults and 18 thousand source-level faults, in a real world system from the space domain, i.e., a satellite data handling system. The proposed method was effective at highlighting the pitfalls that can occur in the implementation of G-SWFIT and affect the accuracy of fault injection. In particular, issues were found in the identification of code blocks and control structures, and in enforcing fault constraints. Moreover, the analysis shows that if identified pitfalls are avoided, the accuracy of G-SWFIT can be significantly improved.

This chapter is structured as follows. Section 4.2 describes the proposed method. Section 4.4 discusses the obtained results by applying the method on a complex system, which is described in Section 4.3. Lessons learned are summarized in Section 4.5.

4.2 Proposed method

As previously discussed, the assessment of binary-level SFI is motivated by the fact that its accuracy is limited by the impossibility to correctly recognize some programming constructs in a binary program. The proposed method is aimed to assess the accuracy binary-level SFI *in the context of real-world complex software*, in order to understand the limitations and the accuracy of the results that can be obtained by a realistic fault injection scenario.

An example of wrongly injected fault is represented by a C program containing a SWITCH construct with two branches; in some architectures and compilers (this is the case of GNU's compiler GCC for PowerPC architectures), it may happen that the SWITCH is translated in binary code using the same opcode sequence of an IF-ELSE construct, since they both consist of a logical condition (which is translated using an opcode that compares two values) and two branches (which are translated using branch opcodes). Therefore, a MIEB (see Table 2.18) fault could erroneously be injected in a code location in which there is not an IF-ELSE construct. It may also happen that a code location suitable for fault injection cannot be recognized in the binary code. For instance, a compiler may translate a function call as inline code (i.e., the function call is replaced with the body of the called function); in this case, a fault injection tool would not be able to recognize the function call, thus omitting to inject an MFC fault in that location. The experimental validation in this chapter aims i) to assess the relative occurrence of this kind of problems in real-world complex software; and ii) to point out issues that may arise when implementing G-SWFIT, by highlighting cases in which faults are not correctly injected. From the previous considerations, it is clear that binary-level fault injection tools are difficult to implement, since they have to encompass all potential ways in which programming constructs are translated. This problem is further exacerbated if we consider the complexity of modern CPUs, programming languages and compilers (whose inner working is usually unknown). Thus it is likely that developers may not correctly implement the handling of some binary code patterns, thus leading to a defective fault injection tool.

The proposed method evaluates the accuracy of G-SWFIT by comparing the faults it generates with the ones injected in the source code. Indeed, since a software fault is a defect in the code of a program, it is clear that fault injection at source code level is more accurate. Based on this consideration, we compare the faults injected by the two techniques and we classify faults in the following three categories:

1. ***Correctly Injected faults***: correct faults generated by both techniques. The larger is the set of common faults, the higher is the accuracy of G-SWFIT.

2. ***Omitted faults***: faults injected only at source-code level. They correspond to programming constructs in which a fault could exist, but which have not been identified in the binary code.

3. ***Spurious faults***: faults injected only by G-SWFIT at binary level that do not match any fault at source-code level. Therefore, they are not considered as representative software faults.

It is important to note that source-level faults can be used as a term of comparison for binary-level faults because (i) *the same fault types are adopted for both binary- and source-level fault injection* (shown in Table 2.18), and (ii) *binary- and source-level faults are injected in every potential location* (i.e., fault injection campaigns are exhaustive). The method (depicted in Figure 4.1) consists of two phases, namely (i) automatic matching of binary-level and source-level faults (Section 4.2.1), in order to identify Correctly Injected faults, and (ii) fault sampling and manual analysis (Section 4.2.2), in order to identify which issues affect the accuracy of G-SWFIT. As a real-world case study, we consider CDMS (Command and Data Management System), a real-time embedded system developed by Critical Software for the space domain, which is described in Section 4.3.

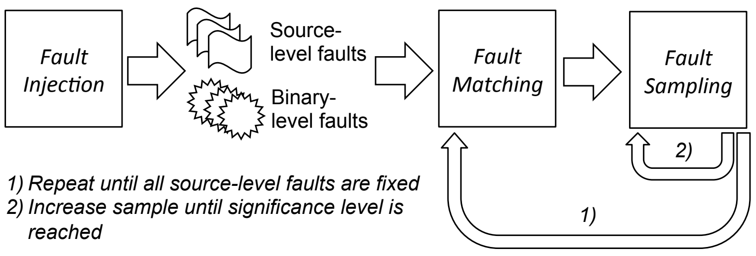
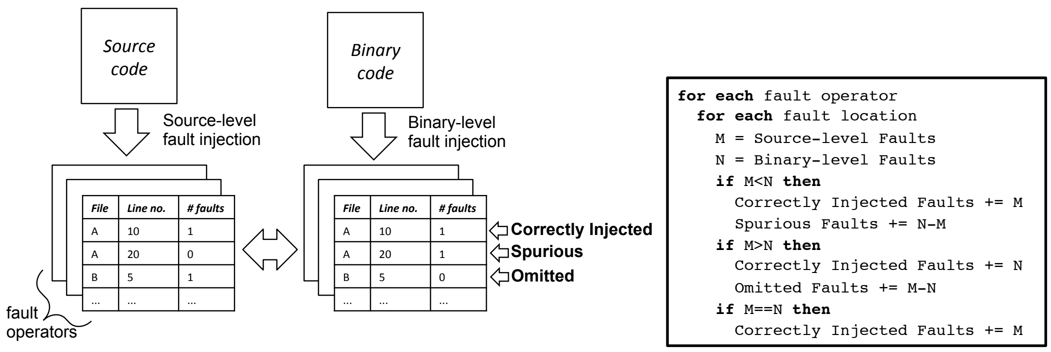

Fault Injection
Source-level faults
Binary-level faults
Fault Matching
Fault Sampling
2)
1)
1) Repeat until all source-level faults are fixed
2) Increase sample until significance level is reached

Figure 4.1: Overview of the method adopted for the evaluation of G-SWFIT.

4.2.1 Fault Matching

Fault Matching is based on the assumption that if both techniques inject the *same fault type* in the *same location* (e.g., an assignment or function call is removed both in the source code and in the corresponding location in the machine code), then they are injecting the same fault. It is reasonable to make this assumption since if a fault location is identified both at the binary and source levels, then that fault location is valid and correctly handled. In order to be sure that this assumption holds (and therefore the results are valid), we manually analyzed a sample of Correctly Injected faults using the Fault Sampling procedure (explained in the next subsection). Following this observation, binary-level and source-level faults are compared with respect to their fault types and their locations in the source code (i.e., the source file, the function and the line of code in which a fault is injected). A binary-level fault matches a source-level fault *if they have the same fault type and they are injected in the same code location.*

The procedure shown in Figure 4.2 has been adopted to identify Correctly Injected faults. If a binary-level fault matches a source-level fault, and only one binary-level fault and only source-level fault exist for the code location under analysis, then the binary-level fault is considered as Correctly Injected. In some cases (e.g., when there are more than one statement in the same line of code), more than one binary-level fault (N), or more than one source-level fault (M) may occur in the same code location. If there are more binary-level faults than source-level faults in the same location ($N > M$), then there are M Correctly Injected faults, and $N - M$ Spurious faults. Similarly, if source-level faults are more than binary level faults ($M > N$), then there are $M - N$ Omitted faults. It follows that if a binary-level fault does not match any source-level fault, then it is considered a Spurious fault, and that if a source-level fault does not match any binary-level fault, then it is considered an Omitted fault. In the examples of Figure 4.2, the proposed procedure identifies one Correctly Injected fault (location *A-10*), one Spurious fault (location *A-20*), and one Omitted fault (location *B-5*).



File	Line no.	# faults
A	10	1
A	20	0
B	5	1
...	...	...

File	Line no.	# faults
A	10	1
A	20	1
B	5	0
...	...	...

```
for each fault operator
  for each fault location
    M = Source-level Faults
    N = Binary-level Faults
    if M<N then
      Correctly Injected Faults += M
      Spurious Faults += N-M
    if M>N then
      Correctly Injected Faults += N
      Omitted Faults += M-N
    if M==N then
      Correctly Injected Faults += M
```

Figure 4.2: Fault matching procedure.

4.2.2 Fault Sampling

After the Fault Matching procedure, we perform a detailed analysis of faults in order to investigate the causes of Spurious and Omitted faults, and to verify that Correctly Injected faults are actually correct. Moreover, we aim to understand whether Omitted and Spurious faults are due to inherent limitations of G-SWFIT or not. Indeed, these faults may occur due to design issues in G-SWFIT as previously discussed; the identification of these issues is useful to provide guidelines for improving G-SWFIT, and to obtain a more precise figure of merit of the G-SWFIT technique. For these reasons, we manually analyze a random sample of Omitted and Spurious faults, and classify them into the following categories:

1. *C preprocessor macros.* When the G-SWFIT technique was proposed, preprocessor

macros have been recognized as a frequent cause of Omitted and Spurious faults [65]. A preprocessor macro consists of a piece of code that is replicated for each time the macro is referred within the program. Therefore, when a preprocessor macro has a software fault, the faulty code is replicated several times in the binary code. Since the binary code lacks information about macros, G-SWFIT cannot recognize that macro code is replicated elsewhere within the program: therefore, a Spurious fault is injected for each replica of the macro, and source-level faults that could be injected into macro represent Omitted faults since G-SWFIT cannot correctly injected them (see also Figure 4.3).

2. *Inline functions.* In a similar way to preprocessor macros, inline functions are replicated each time the function is called within the program. Since G-SWFIT does not recognize inline functions within binary code, they lead to Spurious and Omitted faults as well.

3. *Various causes.* In this category, we include all the other causes of Spurious and Omitted faults that are not related to macros or inline functions.

4. *Issues in the SAFE tool.* Even if source-level fault injection can be considered accurate, we did not exclude this possibility that the source-level fault injection tool we adopt could inject faults incorrectly. Therefore, during the manual analysis, we also look

for issues in the SAFE tool that caused faults to erroneously appear as Spurious or Omitted faults. Since we need to assure that source-level faults are correctly injected, we fix the SAFE tool when an issue is found and repeat the whole analysis (including both Fault Matching and Fault Sampling) until this category becomes empty.

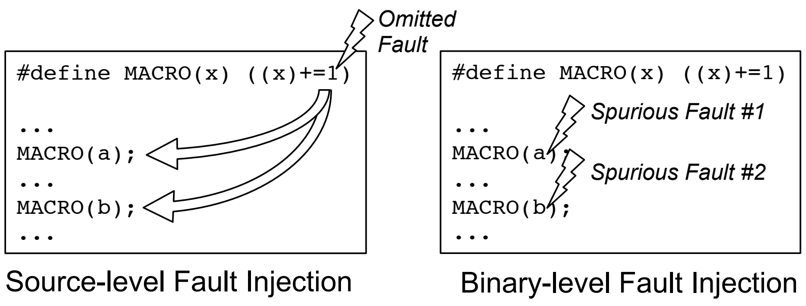

Omitted Fault
#define MACRO(x) ((x)+=1)
...
MACRO(a);
...
MACRO(b);
...
Source-level Fault Injection
#define MACRO(x) ((x)+=1)
...
MACRO(a);
...
MACRO(b);
...
Spurious Fault #1
Spurious Fault #2
Binary-level Fault Injection

Figure 4.3: Example of Spurious and Omitted faults due to the occurrence of a C preprocessor macro within a program.

Because of the high number of the generated faults, the manual analysis is conducted on a sample of faults and then conclusions are drawn about the whole set of faults. In order to generalize the results from the sample, we have to address the problem of choosing a sample of appropriate size, such that it could be considered representative of a population with more than two categories (i.e., a multinomial distribution, where we define π_i as the proportion of the ith category). The sample should be large enough to assure that all of the estimated proportions π_i are within a given confidence interval with significance level $1 - \alpha$.

Assuming that the population and the sample are large enough to use the normal approximation, the probability α_i that the proportion π_i lies outside an interval of width $2d_i$

is given by (see [190] for more details about sampling)

$$\alpha_i = Pr\left\{|Z_i| \geq d_i\sqrt{n}/\sqrt{\pi_i(1-\pi_i)}\right\} \tag{4.1}$$

where $1 \leq i \leq k$ and Z_i is a standard normal random variable. By Bonferroni's inequality [190], the probability that one or more of the k estimates will fall outside its interval will be less than or equal to $\sum_i^k \alpha_i$. Equation (4.1) allows to assess if the sample size is large enough to achieve accurate results. If $\sum_i^k \alpha_i > \alpha$, then a larger sample size is required, otherwise the estimated proportions are considered accurate.

This method was applied to the populations of Omitted and Spurious faults by considering $k = 4$ categories (C preprocessor macros, inline functions, various causes, issues in the SAFE tool), assuming a confidence interval of half-width $d_i = 0.05$ and a significance level $1 - \alpha = 0.9$. This method was also applied to the population of Correctly Injected faults, in order to analyze whether they are truly correct or not ($k = 2$ categories are considered). For each population, we extract a sample of 5% of faults and then we manually analyze each fault in order to obtain an initial estimate of the proportions; the sample size is gradually increased and analyzed until the required significance level is reached. The results are described in the Section 4.4.

4.3 Case study

The case study considered in this chapter is a satellite data handling system named Command and Data Management System (CDMS). A satellite data handling system is responsible for managing all data transactions (both scientific and satellite control) between a ground system and a spacecraft (Figure 4.4), based on the ECSS-E-70-41A standard [69] adopted by the European Space Agency. In this system, a space telescope is being controlled and the data collected is sent to a ground system. As shown in the Figure, the CDMS, which executes on the spacecraft (*on-board system*), is composed by several subsystems: the TC Manager receives a series of commands from the ground control requesting telemetry information; the TM Manager sends back telemetry information for each command sent; the other modules (PC, PL, OBS, RM, DHS) perform tasks for the data management and the telescope handling. The importance of the *accuracy* of SFI in mission-critical systems like CDMS has been demonstrated in [143], in which two OSs (RTLinux and RTEMS) were compared with respect to the risk of failures of the CDMS due to OS faults, in order to select the most reliable OS for this scenario.

The CDMS application was developed in C and runs on top of an open-source, real-time operating system, namely RTEMS[1]. The CDMS makes use of the RTEMS API for task management, communication and synchronization, and for time management. This software

[1]`http://www.rtems.org`

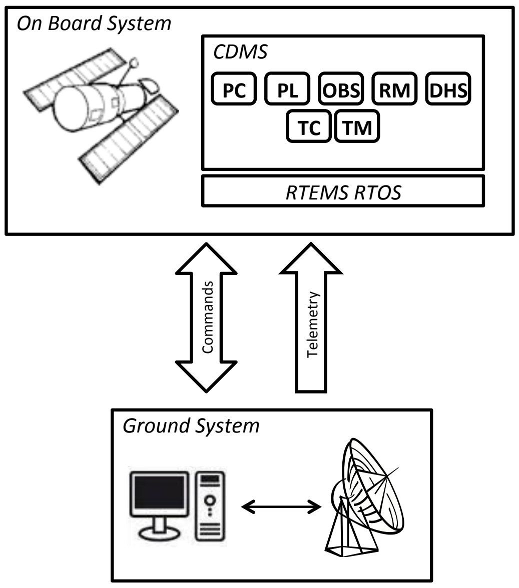


Figure 4.4: Architecture of the case study.

system is compiled to run on a PowerPC hardware board by using the GCC compiler and disabling compiler optimization settings, which is the setup currently supported by the G-SWFIT tool.

In this case study, faults are injected in both the OS (i.e., RTEMS) and application (i.e., CDMS) code. *We only consider the code which is actually compiled and linked in the executable running on the on-board system.* A small part of the code (1.90%), which is written in assembly language to provide board-specific support, is not targeted by our source-level fault injection tool, but its influence on the results can be considered negligible.

4.4 Results

In this section, software faults injected at the binary and source-level in a complex case study are analyzed using the method proposed in Section 4.2. Faults at the binary level were generated with the G-SWFIT technique, by using a R&D prototype tool provided by Critical Software [181]. Faults at the source code level were generated using the SAFE fault injection tool (described in Subsection 3.2.2). In total, 18,183 source-level faults and 12,380 binary-level faults were generated, respectively. Their distribution across fault types is shown in Figure 4.5. The two distributions exhibit noticeable differences: more binary-level faults are injected with respect to some fault operators (such as OMLPA, OWVAV, OWPFV, and OWAEP), whereas in other cases more source-level faults are injected (such as OMIEB and OMVA, where the latter groups together the OMVAV, OMVIV, and OMVAE operators).

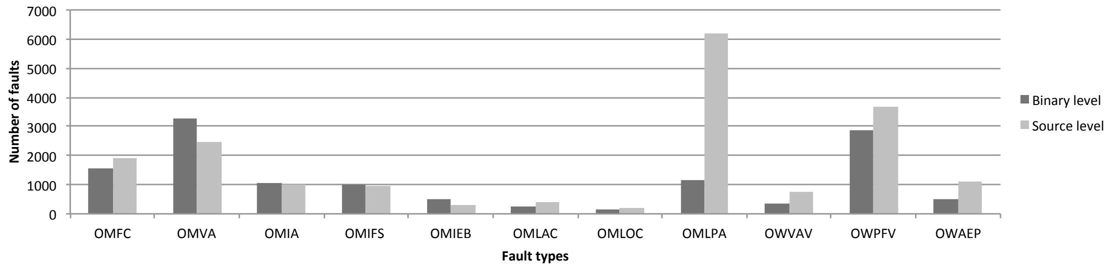


Figure 4.5: Distributions of software faults injected at the binary and source code level, respectively.

The Fault Matching procedure (Section 4.2.1) identified the subset of Correctly Injected faults (i.e., common to both techniques), which we further analyzed in order to assure the correctness of our method. Correctly Injected faults have been sampled (see Section 4.2.2), and then analyzed by comparing i) the faulty binary-code generated by G-SWFIT, and ii) the one produced by faults injected in the corresponding source-code location. This analysis revealed that the binary-level faults match the source-level faults for each fault type and for each sampled fault, except the OWPFV operator. We found that 40.69% of OWPFV faults at the binary level do not match OWPFV faults at the source-code level even if they affect the same locations, since there are several functions parameters and possible replacements for a given location. In order to take into account this aspect, results shown in Figure 4.6 have been updated by reducing the number of Correctly Injected faults for the OWPFV operator and increasing the number of Omitted and Spurious faults by the same amount.

Correctly Injected faults turned out to be 5,927 (Figure 4.6). They represent 47.88% of faults injected by G-SWFIT. The remaining faults injected by G-SWFIT (52.12%) in the binary code do not match to a software fault in the source code, therefore most of G-SWFIT faults are Spurious. Correctly injected faults represent 32.60% of faults injected in the source code, so the remaining faults at the source level (67.40%) are not emulated by G-SWFIT and they result as Omitted faults. The experimental campaign confirms that the accurate injection at the binary level is a challenging task, at least when a complex software system is considered.

The distribution of the causes of inaccuracies (for both Omitted and Spurious faults) are

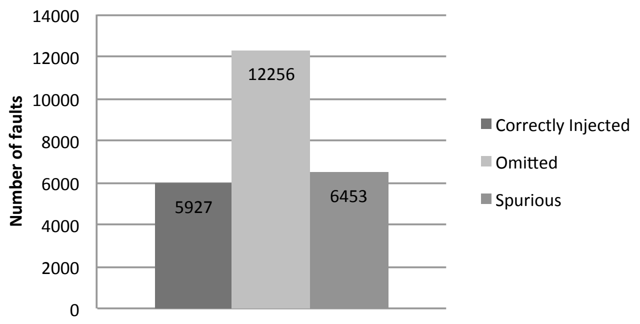
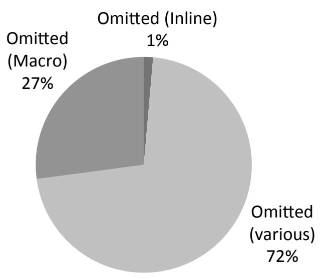


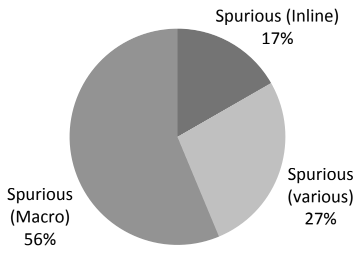
Figure 4.6: Number of Correctly Injected, Spurious, and Omitted faults.

presented in Figure 4.7. These distributions have been obtained by applying the sampling procedure described in Section 4.2.2. Most of spurious faults (Figure 4.7b) are caused by C macros (56%) and inline functions (17%). In these cases, every time that a macro or inline function has been replicated in the binary code, G-SWFIT generated an individual binary-level fault; this led to a large number of Spurious faults (i.e., Spurious faults are repeated for each replica of a macro or inline function). In a similar way, macros and inline functions are a noticeable part of Omitted faults (27% and 1%, respectively); this percentage is low when compared to Spurious faults, since one Omitted fault in a macro or inline function leads to several Spurious faults, one for each replica of the code (see also Figure 4.3).

In order to gain more insights into the results, we separately analyzed the faults injected in the OS and application code, respectively. Figures 4.8 and 4.9 show from a different perspective the data of Figures 4.6 and 4.7, by dividing the results between faults in RTEMS (i.e., OS code) and in CDMS (i.e., application code). It can be noted that faults follow a



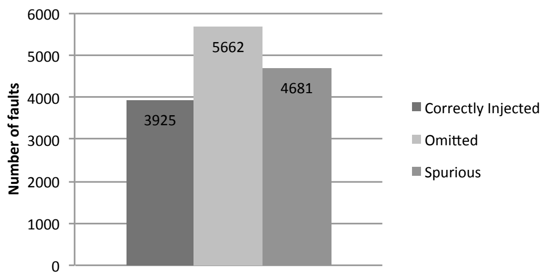
(a) Omitted faults.



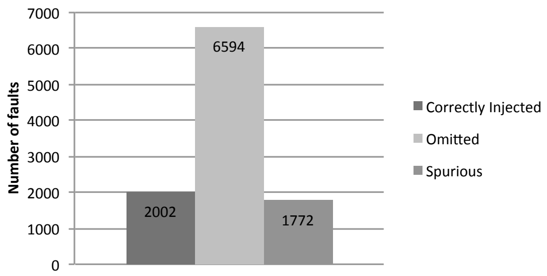
(b) Spurious faults.

Figure 4.7: Causes of incorrect fault injection in the case study.

similar trend in OS and application code, since in both cases the number of spurious faults is close to the number of correctly injected faults, and the number of omitted faults is predominant. Nevertheless, omitted faults seem to be much more in the case of CDMS (Figure 4.8b).



(a) RTEMS code.

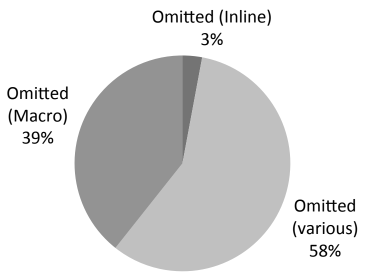


(b) CDMS code.

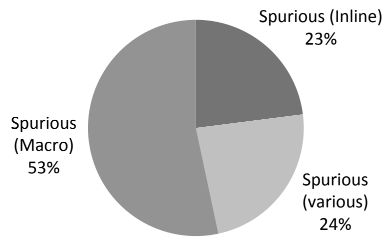
Figure 4.8: Number of faults (correctly injected, spurious, and omitted) in OS and application code.

Figure 4.9 shows that omitted and spurious faults due to various causes (i.e., not related to macro or inline functions) are more frequent in CDMS than in RTEMS. The constructs not correctly recognized at the binary level (e.g., see the examples in Figures 4.11 and

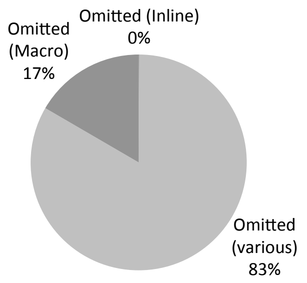
4.12 discussed later in this section) likely occur more often in application code due to higher complexity of that code, thus causing an higher number of omitted faults. Moreover, macros and inline functions are more frequent for RTEMS; this is due to the fact that several RTEMS functions are exported as macros and inline functions in order to be used by external code (i.e., user and library code that is compiled and linked with RTEMS code).



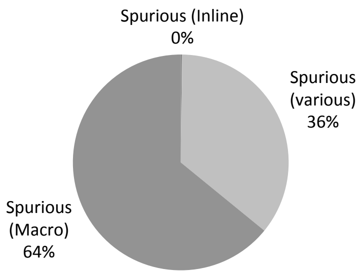
(a) Omitted faults in RTEMS.



(b) Spurious faults in RTEMS.



(c) Omitted faults in CDMS.



(d) Spurious faults in CDMS.

Figure 4.9: Causes of incorrect fault injection in OS and application code.

Software complexity metrics collected from the case study code (see Table 4.1) confirm that functions in the application code tend to be more complex than those in the OS code (in term of size, cyclomatic complexity and input/output dependencies). This is a common trend in embedded systems, in which the OS is kept as simple as possible in order to reduce the overhead and the number of potential defects [165]. Moreover, the number of preprocessor statements per function confirms that RTEMS makes a more extensive use of macros that CDMS. Therefore, we conclude that it is even more important to fix the implementation issues mentioned above if a fault injection tool is intended to be used with complex software.

Table 4.1: Comparison of average software complexity metrics of functions in RTEMS and CDMS code.

Metric	**RTEMS**	**CDMS**
Lines of Code	17.30	30.71
Preprocessor Statements	0.64	0.15
Cyclomatic number	5.63	6.61
Number of inputs	5.50	7.38
Number of outputs	4.12	6.84

The "various causes" behind spurious and omitted faults are numerous and specific to each fault operator. We cannot provide a precise estimate of the relative percentage of each cause, since it would require to manually analyze an extremely large sample of injected faults. Instead, we tried to identify which part of incorrectly injected faults are due to unavoidable limitations of G-SWFIT, and which of them can be avoided by improving the G-SWFIT fault injection tool. We do so by excluding from the sample those faults not related to macros or inline functions, and by diagnosing (with the support of Critical Software developers) the reasons why omitted faults were not injected, and why spurious faults were erroneously injected. We found that 26.02% of omitted and spurious faults were due to causes that are impossible to avoid when injecting at the binary code level, including:

- *Low-level translation of C operators.* Some C expressions (like *sizeof* and array and struct accesses using -> and *[]*) are translated by introducing arithmetic operations and constants in the binary code; these operations are recognized as arithmetic expressions by fault operators such as OMVA, OWVAV, and OWAEP.
- *Switch and goto constructs.* These constructs are translated in a similar way to IF constructs using *branches* in the binary code; therefore, IF constructs are not always correctly identified by operators such as OMIA, OMIEB, and OMIFS.
- *Forced function inlining.* Some functions (e.g., *memcpy*, *memset*) are compiled as inline functions, although they are not declared as inline.

Since the binary code lacks information about high-level constructs, the causes mentioned above cannot be avoided. In practice, these inaccuracies have to be accepted as limitations of fault injection at binary level, and should be taken into account when conclusions are drawn from fault injection experiments.

Nevertheless, during the manual analysis we observed several Omitted and Spurious faults not related to intrinsic limitations of fault injection at binary level, but were due to limitations of the fault injection tool; they represent the 73.98% of the sample that we analyzed. These inaccuracies occurred since some checks have not been implemented yet in the tool, and some fault operators diverge in some cases from the fault types encompassed by G-SWFIT due to choices that simplify the implementation. Therefore, part of the Omitted and Spurious faults could be avoided by improving the implementation of binary-level fault injection.

An example of Spurious fault is provided in Figure 4.10, which shows a fault location in the source code (monospace font) along with its machine code translation (italic font). It is a spurious MFC fault in CDMS that has been injected in a wrong location. In this example, the function call should not be removed since it is the only statement within a block of code, and a fault in that location would not be realistic. The OMFC operator imposes a constraint (Table 2.19) to avoid fault injection in this kind of location [65]. Instead, the fault has been injected by the tool since the block containing the function call is not recognized (i.e., the constraint is not enforced by the tool).

Figure 4.11 and Figure 4.12 provide two examples of Omitted faults that were caused by limitations in G-SWFIT implementation. In Figure 4.11, a function call which could be removed by the OMFC fault operator is not identified. As confirmed by Critical Software developers, the *TcMakePacket* function is not recognized as returning a value that is stored and used later in the program. Therefore, a fault is not injected due to a constraint of the OMFC operator requiring that the return value of a function should not be in use (Table

```
static void HousekeepingAction(TmPacket *STm) {
        stwu r1,-24(r1)
        mflr r0
        stw r31,20(r1)
        stw r0,28(r1)
        mr r31,r1
        stw r3,8(r31)

   SendTmMsg(pbtBuffer,
      TmGetPacketTotalLength(STm));  ← MFC fault location
        lwz r3,8(r31)                  (to be avoided)
        bl 00006184 <TmGetPacketTotalLength>
        mr r0,r3
        lis r9,7
        addi r3,r9,-21944
        mr r4,r0
        bl 0000a3b4 <SendTmMsg>

}
        lwz r11,0(r1)
        lwz r0,4(r11)
        mtlr r0
        lwz r31,-4(r11)
        mr r1,r11
        blr
```

Figure 4.10: Spurious MFC fault in CDMS.

2.19).

```
    TcMakePacket(pbtBuffer, &STc);  ← MFC fault location
        addi r0,r31,24                 (not identified)
        lis r9,9
        addi r3,r9,-21492
        mr r4,r0
        bl 000056b8 <TcMakePacket>

    bOk = CheckAppIdTypeSubtype(&STc);
        addi r0,r31,24
        mr r3,r0
        bl 00011a10 <CheckAppIdTypeSubtype>
        mr r0,r3
        stw r0,20(r31)
```

Figure 4.11: Omitted MFC fault in CDMS.

In Figure 4.12, the fault location has been omitted for an even more subtle reason. In this example, the *return* statement within the IF construct is translated with a branch to the end of function, and the tool incorrectly believes that the IF construct includes all the statements until the end of the current function. A fault is not injected since the IF construct should not contain more than 5 statements [65]. Although the tool is provided with checks to avoid these mistakes, a check to avoid this specific case was not implemented. This kind of issue seems to be more relevant for Omitted faults than for spurious faults given the high number of omitted faults due to various causes, as depicted in Figures 4.6 and 4.7.

```
rtems_status_code sc;
n32Size = TcGetAppData(STc, &pbtData);
    lwz r3,120(r31)
    lis r9,7
    addi r4,r9,-23004
    bl 00005934 <TcGetAppData>
    mr r0,r3
    lis r9,7
    stw r0,-22992(r9)

sc = rtems_semaphore_obtain(rtems_mon_Mutex,
                            RTEMS_WAIT,
                            RTEMS_NO_TIMEOUT);
    lis r9,7
    lwz r0,-22948(r9)
    mr r3,r0
    li r4,0
    li r5,0
    bl 0003d504 <rtems_semaphore_obtain>
    mr r0,r3
    stw r0,64(r31)

if (sc != RTEMS_SUCCESSFUL)      ← MIA fault location
    lwz r0,64(r31)                 (not identified)
    cmpwi cr7,r0,0
    bne- cr7,0000c69c <AddMonitoringAction+0x97c>
    return;

if ( n32Size >= 10 ) {
    lis r9,7
    lwz r0,-22992(r9)
    cmplwi cr7,r0,9
    ble- cr7,0000c680 <AddMonitoringAction+0x960>
```

Figure 4.12: Omitted MIA fault in CDMS.

Other incorrect behaviors were also found in the prototype tool, which were due to the incomplete implementation of contraints or the identification of code blocks and control structures. In Figure 4.13, we provide an evaluation of the results that can be obtained by improving the mentioned aspects. The improvements prevent the occurrence of several

Omitted and Spurious faults: Correctly Injected faults represent the majority of faults potentially injectable in the source code (i.e., only a minor part of faults is omitted), and they also represent the majority of faults actually injected by G-SWFIT (i.e., only a minor part of faults is spurious). We conclude that the evaluation of a binary-level fault injection tool on real-world complex software is useful to identify implementation issues, and should be adopted to assure that a tool does not omit valid fault locations, and that spurious faults are not generated.

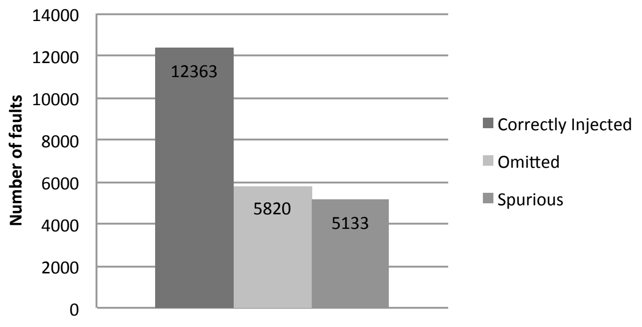


Figure 4.13: Number of faults (correctly injected, spurious, and omitted) when fixing implementation issues of the G-SWFIT tool.

4.5 Summary

In this chapter, we evaluated the accuracy of a software fault injection technique (G-SWFIT) that injects faults in the binary code of a program. The accuracy of faults injected at binary level has been assessed by comparing the faults injected in the source code by using the same fault injection rules. The analysis pointed out improvements to both tools involved in the comparison. Results can be summarized as follows:

- The accurate injection of software faults in the binary code is challenging in complex software systems. A large number of omitted and spurious faults was observed in the first analysis: for each injected fault there is about 1 omitted fault that has not been injected, and about half of the injected faults were spurious. Moreover, the problem is more significant where the code complexity is greater, as in the case of application-level code in the case study.

- Several omitted and spurious faults are due to the lack of high-level information in the binary code, and most of them are due to macros and inline functions. These inaccuracies have to be accepted as limitations of fault injection at binary level, and should be taken into account when conclusions are drawn from fault injection experiments. In some cases, such limitations can be considered acceptable: for instance, when the aim of fault injection is a coarse-grained analysis of failure modes (e.g., the relative percentage of crashes or stalls of the system), the results can be adequately estimated even in the presence of inaccurate injected faults [65, 107]. Instead, fault injection at the source level is advisable when the source code is available and a more fine-grained analysis of the effects of injected faults on the system is needed.

- Several omitted and spurious faults are not related to limitations of fault injection at binary level, but they are due to the incomplete or simplified implementation of G-SWFIT. In particular, issues are related to the implementation of constraints in fault operators and the identification of code blocks and control structures. These issues are not due to the G-SWFIT technique, and they can be avoided if an experimental evaluation of the fault injection tool is performed to improve the implementation. If these aspects are improved, then omitted and spurious faults represent the minority of cases. A future research direction consists in extending the proposed method in order to support the development of software fault injection tools at binary level, since such tools need to be re-engineered or developed from scratch when fault injection is performed in a new hardware architecture or in a system adopting a different compiler. To this aim, faults injected at the source code level can be potentially exploited to understand how software faults are translated in binary code and how fault operators can be implemented.

Chapter 5

A technique for the emulation of concurrency faults

5.1 Introduction

In the last decades, several studies on software systems contributed to better understand software faults and how to deal with them [83, 126, 187, 128, 34, 86]. As these studies confirmed, dealing with software faults is a difficult task: the main problem is the *reproducibility* of the failure, that is, the ability to identify the conditions that make the fault activate, which is required to track down and fix a fault. Faults whose activation is reproducible are called *Bohrbugs*. They are typically detected and then fixed during testing phases. *Mandelbugs*, instead, are faults whose activation is not systematically reproducible and they typically lead to transient failure manifestations [88]. Their activation conditions (namely, *fault triggers*) depend on complex combinations of user inputs, the internal state, and the external environment, i.e., the set composed by other programs, services, libraries, virtual machines, middleware and operating system the application interact with. The activation conditions of Mandelbugs (e.g., a thread schedule that triggers a concurrency fault) can be very difficult to reproduce. For this reason, testing activities reveal to be not effective for dealing with such a kind of faults. As a result, Mandelbugs manifest themselves only during the operational phase, and account for a significant part of failures in critical software systems [126, 88, 86].

Although it is almost unfeasible to avoid the occurrence of Mandelbugs, their transient behavior makes it possible to mask them by retrying the failed operation [166] (*temporal redundancy*), or by switching to a backup process [126, 135] (*spatial redundancy*). These fault tolerance approaches are effective against Mandelbugs since their activation conditions are complex and rare, and thus they are unlikely to occur again when the operation is retried.

Despite the relevance of Mandelbugs and of fault tolerance mechanisms used to deal with them, the emulation of these faults was overlooked by Software Fault Injection studies, due to the lack of data about these faults (since they are difficult to reproduce, it is also difficult for user and developers to diagnose and to document them), and to their complex nature and interactions with the environment (scheduling, timing of events, and interactions with the system state) that are difficult to be emulated through SFI. For instance, G-SWFIT deliberately excludes the ODC fault type Timing/Synchronization [65]. Other studies, such as the ones by Ng et al. [151, 152, 153], adopt a simplistic approach to mimic these faults, for instance by causing the procedures that acquire/free a lock to return without acquiring/freeing the lock: this approach is not efficient since it relies on the random occurrence of conditions that make these faults to activate (in this case, a specific thread schedule). This problem is not easy to be solved, since emulation of Mandelbugs, in addition to fault injection into the executable code, also requires the emulation of fault triggers that activate them.

In this chapter, **a technique is proposed for the injection of *concurrency faults* in multithreaded software**. The technique emulates these faults in a fully representative way, by controlling the environment conditions that make these faults to activate and manifest themselves. We focus on concurrency faults as they represent a critical sub-class of Mandelbugs, because (i) they are the most frequent and severe kind of Mandelbugs [187, 128, 86], and (ii) we are experiencing a shift towards multithreaded software and multicore hardware architectures, which are prone to this kind of faults [56]. The representativeness of emulated faults is assured by a field data study on real concurrency faults observed in the field, and by the precise emulation of their activation conditions using a technique specifically designed for this purpose.

First, we analyze the limitations of the state-of-the-art technique G-SWFIT regarding its ability to emulate the transient nature of Mandelbugs, in the context of a fault-tolerant software system for Air Traffic Control. It is found that injected faults do not realistically emulate Mandelbugs, since most of them are activated in the early phases of execution, and they deterministically affect process replicas in the system. Moreover, this behavior impacts on the verification of fault tolerance, as 35% of system states are not covered during the fault injection campaign. Subsequently, by means of an experiment, we show how concurrency faults can be emulated in the same system in a representative way. The emulation of concurrency faults reveals to be useful for improving the assessment of fault tolerance mechanisms, since controlling fault activation can reduce the amount of untested states down to 5% and therefore increase the confidence in fault tolerance.

This chapter is organized as follows. Section 5.2 describes the case study considered in this chapter. In section 5.3, G-SWFIT is analyzed. In section 5.4, we describe a technique for emulating concurrency faults, which is evaluated in section 5.5. Section 5.6 summarizes the chapter.

5.2 Case study

The case study considered in this chapter consists of a mission-critical system for ATC, namely Flight Data Processor System (FDPS), and the middleware on which it is based, namely CARDAMOM [44]. This section describes this system, with emphasis on the fault tolerance mechanisms that are adopted. This system will be considered for analyzing both G-SWFIT and the proposed technique, with respect to the representativeness of SFI and its effectiveness for assessing fault tolerance.

5.2.1 CARDAMOM middleware

CARDAMOM is a middleware platform that provides features to configure, deploy and execute near real-time, distributed and fault-tolerant applications [44]. It is a CORBA-based, OMG compliant platform supporting both the object- and the component-based programming models.

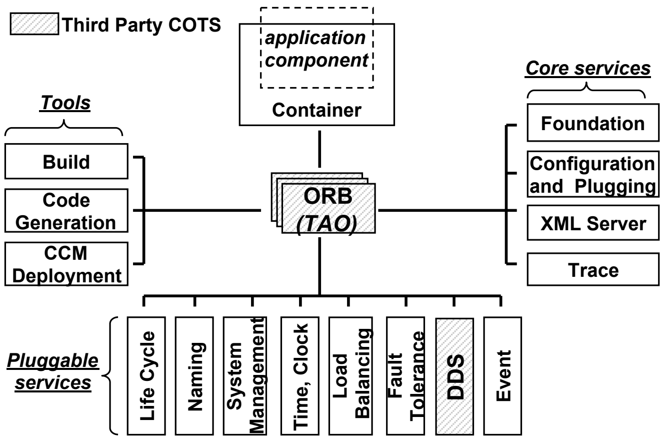


Figure 5.1: CARDAMOM overview.

The high-level architecture of the middleware is sketched in Figure 5.1. Its features are provided in the form of support tools and services. The former aim to simplify the task of application development (e.g., automatic code generation tools). The latter are further divided in core (i.e., mandatory) and pluggable (i.e., optional) services.

CARDAMOM relies on several off-the-shelf components; in particular, it makes use of the Linux operating system, and the ACE ORB (TAO) [98]. Moreover, the platform includes an implementation of the OMG Data Distribution Service (DDS) standard for publish-subscribe communication, namely RTI DDS [99]. DDS enables data sharing among distributed processes, without concern for their actual physical deployment.

Two CARDAMOM services are relevant in the context of this thesis: the *Fault Tolerance* (FT) and *Load Balancing* (LB) services. The FT Service is compliant to the FT CORBA Specification. It provides redundancy by means of CORBA object replication: when a faulty primary replica is detected (e.g., a replica is terminated unexpectedly), the FT Service elects a new primary replica from a pool (namely *object group*). CARDAMOM implements the *warm passive* replication schema, i.e., when the state of the primary replica is modified, it gets recorded and transferred to other members in the object group. The fault-tolerant application is responsible for maintaining consistency between replicas, by means of an API provided by CARDAMOM.

The LB Service allows the distribution of CORBA requests among the members of an object group. A request is redirected to one of the servers, according to an user-defined policy (e.g., Round-Robin, Random). The LB Service is transparent from the client point of view.

5.2.2 FDPS

FDPS is a C++ application based on CARDAMOM. It represents the part of an ATC system in charge of managing Flight Data Plans (FDPs). An FDP is a data structure containing information about a flight, and the goal of FDPS is to keep FDPs up-to-date with respect to the flight status. For example, FDPS has to analyze the actual position of aircrafts (retrieved from radar tracks) and update flight routes consequently, in order to efficiently allocate the flight space and to avoid flight collisions. A simplified view of FDPS architecture is shown in Figure 5.2.

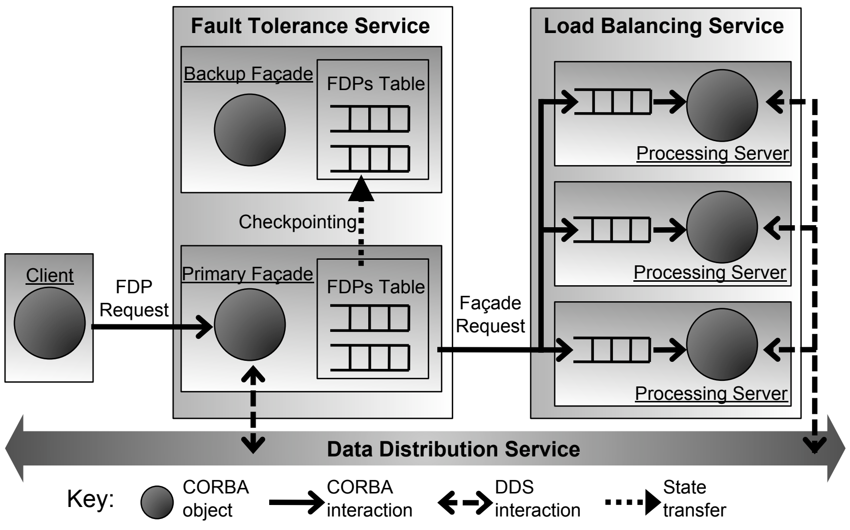


Figure 5.2: Simplified architecture of FDPS.

FDPS is composed by a Façade component, replicated by the FT Service, and a set of Processing Servers (PSs), managed by the LB Service. The Façade is in charge of interfacing the FDPS with external systems (e.g., a graphical user client), and to manage the state of FDPs. When the Façade receives an FDP request, it locks the FDP (at most one request at a time for the same FDP can be processed) in an internal data structure (namely, the FDPs table), and it sends a processing request to a PS. The PS retrieves the FDP and radar tracks from the DDS, processes the FDP, and returns the FDP to the Façade. The Façade then updates the FDP on DDS and unlocks it. If several requests are sent to the same PS, they are executed one at a time. The Façade is responsible for queueing concurrent requests for the same FDP, and for checkpointing the FDPs table. The most important FDP operations, that we take into account, are the insertion, deletion, and update of an FDP.

5.3 Evaluation of G-SWFIT

In this section, we investigate whether faults injected by G-SWFIT are representative of Mandelbugs. In order to better understand the behavior of injected faults, the analysis is supported by a Finite State Machine (FSM) model of the system, which is described in Subsection 5.3.1. Subsections 5.3.2 and 5.3.3 describe respectively the experimental setup and results.

5.3.1 Modeling FDPS

In this section, a Finite State Machine model of the FDPS is presented. This model is used to analyze the system behavior in the presence of injected faults, which is compared to the expected behavior that would be induced by Mandelbugs. The model takes into account the different states in which the system operates, since it is well-known that the ability of the system to correctly handle a fault is dependent on the system state in which it is operating [32, 8, 109]. Moreover, the current system state contributes to the activation of Mandelbugs, along with interactions with the environment (e.g., external events and thread scheduling).

For these reasons, it is important for injected faults to cover as much states as possible, in order to assess fault-tolerance in the several conditions in which faults can manifest and in which the system operates.

In a FSM, each state represents a different combination of internal system variables. States are connected by transitions, which represent events (e.g., an external input) that change the value of internal variables. When the state is changed, the system performs a computation in answer to the occurred event.

The choice of internal variables is a trade-off between the level of detail and the complexity of the model. A too accurate model suffers from the explosion of the number of states, which makes the analysis unfeasible. Therefore, we select a proper subset of system variables and of their possible values, in order to keep the number of states low, and to still take into account the features of the system most relevant to fault tolerance testing. The considered variables are:

- $\#QF$: Number of FDP requests queued by the Façade;
- $\#UP$: Number of Façade requests under processing;
- $\#QP$: Number of Façade requests queued by PSs.

Moreover, we consider messages exchanged within the FDPS (shown in Table 5.1) as transition events, because they are easy to be collected, and they enable to track the values took by the considered variables.

Table 5.1: Messages exchanged within the FDPS.

Name	**Description**
CR	A Client request for an FDP not already requested
CRQ	A Client request for an FDP already requested
FR	A Façade request for an FDP request not in the Façade queue
FRQ	A Façade request for an FDP request in the Façade queue
PSC	A PS returns an FDP, and no other Façade requests are queued by the PS
PSCQ	A PS returns an FDP, and there are Façade requests queued by the PS

To make the model finite, the number of requests that can be queued by the Façade is bounded ($\#QF \leq 3$), without any loss of generality. For the same reason, we also choose to assign only two possible values to $\#UP$, respectively the absence of requests queued by PSs ($\#UP = 0$), and the presence of one or more requests queued ($\#UP = 1$). Instead, the number of requests under processing is bounded by the number of PSs ($\#UP \leq 3$ in the current FDPS architecture). The chosen internal variables and events do not take into account which particular FDPs or FDP operations are under processing, but only how many FDPs or FDP operations are involved, and whether a FDP operation involves an already queued FDP (CRQ, FRQ, PSCQ) or not (CR, FR, PSC). This choice reduces the number of states from several thousands to 20 (Figure 5.3).

5.3.2 Experimental setup

In order to study the manifestation of injected faults with respect to system states, the application has been instrumented to log the contents of input and output messages of the

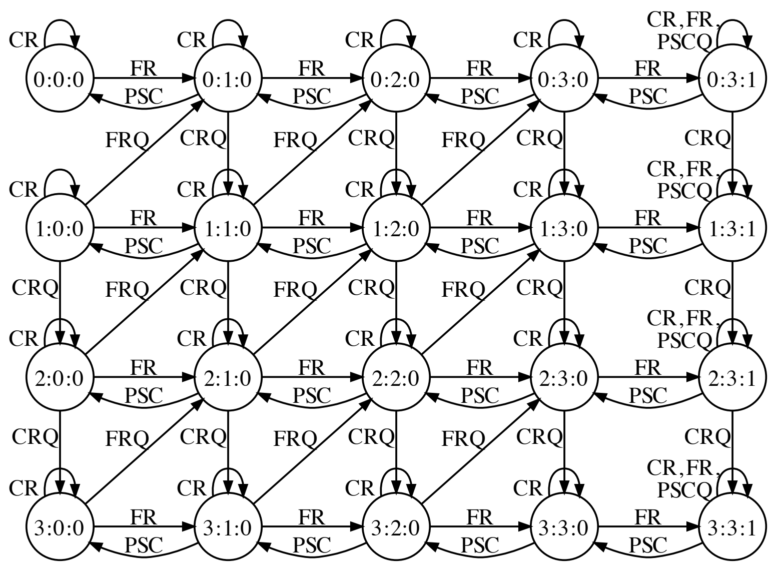


Figure 5.3: A Finite State Machine that models the FDPS.

Façade (Table 5.1). Moreover, a log message is produced before a faulty piece of code is executed. By analyzing these logs after a fault injection experiment, it is possible to trace which states were reached during an experiment. A failure has occurred if the sequence of Façade states does not match the state sequence of faulty-free runs.

Regarding the workload used for SFI experiments, we designed a set of 3 workloads. Each workload makes the system to reach all states in the FSM (starting from *0:0:0*). The workloads differ in the direction in which the FSM is visited (e.g., row-by-row, column-by-column) and the type of the requests (insert, update, delete).

As for the faultload, fault operators encompassed by G-SWFIT (Table 2.18) are used to inject faults into the business logic of the Façade. Fault injection is focused on the

Façade, since it is the most complex component in the FDPS, and therefore the most fault-prone [143]. Moreover, it represents a fundamental entity in the FDPS architecture. Faults were injected using the SAFE fault injection tool (described in Subsection 3.2.2). The experimental campaign encompasses 533 faults, and 1599 fault injection experiments; in 521 cases we noticed a failure of the primary Façade.

5.3.3 Result discussion

For each experiment in which a failure occurs, we consider the state of the system before the failure, and the state in which the faulty piece of code is executed before the failure. Figure 5.4 shows the distributions of these events across system states. A detailed analysis of these distributions reveals that:

- A great amount of faults (55.85%) manifest themselves before the Façade is ready to receive and to process requests, or when the Façade receives the first request (state *0:0:0*). Although these faults are useful to test fault tolerance during the initialization phase of FDPS, they are not well representative of Mandelbugs, which can unexpectedly manifest themselves in any state during the operational phase of a system.

- Faults injected by G-SWFIT are still useful to test fault tolerance with respect to a subset of important system states. In particular, faults that manifest when at least one request is queued by Façade ($\#QF > 0$) allow testing of the checkpointing mechanism (i.e., whether the FDPs table is correctly sent to the backup Façade) and failure detection mechanisms.

- In most of cases (93.3%) in which the backup Façade is activated, the fault causes the failure also of the replica. This suggests us that injected faults are activated deterministically. In these cases, the fault activation is simple to reproduce (as in the case of Bohrbugs). This result biases evaluation of FTMs, which should instead focus on Mandelbugs in well-tested critical software.
- A significant part of states (35%) is not covered by the SFI campaign. Moreover, there are some states in which the percentage of activated faults is very low (i.e., *1:2:0*, *2:2:0*, *3:3:0*), or the faults are activated only under 1 out of 3 workloads (i.e., *3:1:0*, *0:3:1*). Thus, these states are not well tested, since it is unlikely that they would be covered under a different workload. The same holds true for those states that were not covered.
- Part of the injected faults (15.36%, Figure 5.5) causes a failure only under 1 workload, and a significant part of faults (56.93%) do not cause a failure. Therefore, even if several of the injected faults could be representative of Mandelbugs, they are seldom (or never) activated due to the missing occurrence of their activation conditions.

Although these results are influenced by the particular FSM model and workloads, we can conclude that G-SWFIT alone does not easily allow testing of FTMs under some specific

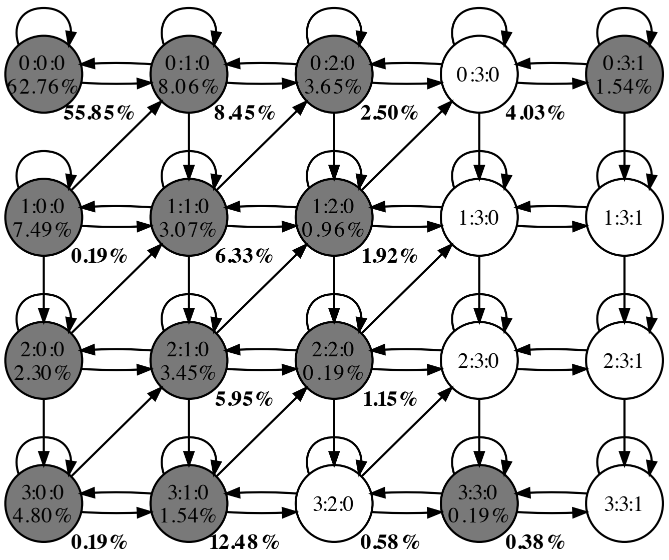


Figure 5.4: Failure and fault activation distributions of the G-SWFIT campaign. The percentage of failures for each state is shown in bold; the percentage of faults activated in each state is shown within the node.

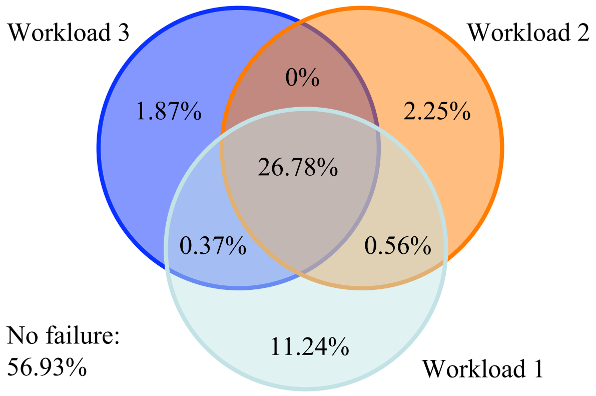


Figure 5.5: Distribution of faults leading to failures (230 out of 534) across workloads. Overlapping areas are faults activated by more than 1 workload.

states. In particular, no fault manifests itself when one or more requests are queued by the PSs ($\#QP > 0$). This fact prevents testing of whether the system correctly tolerates a failure in this scenario. We believe that the problem is due to the lack of control on the fault activation. This result justifies the study of whether faults can manifest in the remaining states, and how they can be activated.

5.4 Concurrency fault injection

The factors that lead to a transient manifestation of Mandelbugs, and the related nature of faults, can be identified by analyzing the scientific literature on field data about software faults. Results of field data analysis guided us to the design of a SFI technique which able to precisely emulate the manifestation of faults, by taking into account the fault activation process.

5.4.1 Fault model

From past works on field data [187, 126, 34, 131], we identified the following transient fault triggers:

- concurrency;
- timing of external events (e.g., from hardware);
- wrong memory state;
- faulty error handling routines (the fault is triggered by another one);

- complex input sequences;
- software aging (e.g., resource leaks).

As discussed in Section 5.1, we focus on faults related to concurrent programming. In order to emulate the most frequent concurrency faults, we take into account the results of a study on concurrency faults [131], which analyzed 105 faults from 4 real complex and concurrent systems. This study pointed out that the most frequent synchronization faults are:

- *atomicity-violation faults* (48.57%), i.e., non-atomic execution of concurrent memory accesses;
- *deadlock faults* (29.52%);
- *order-violation faults* (22.86%), i.e., execution of a set of operations in the wrong order[1].

In particular, we focus on atomicity-violation faults, since they occur most frequently. Moreover:

- non-deadlock faults involved one variable (66.22%);
- 2 threads are needed for triggering a fault (89.52%).

[1]2.86% of the faults are classified as both atomicity- and order-violation.

There are several strategies to enforce the atomicity of a group of memory accesses. However, it is well known that the use of *locks* is the most common strategy used by programmers. Atomicity-violation faults are fixed by programmers using lock primitives, i.e., by adding lock operations before and after accessing shared variables. Although there also exist other fix strategies, the field study [131] points out that they are mostly applied in the case order-violation bugs. Since we focus on atomicity-violation faults, we will consider faults related to the missing usage of lock primitives.

To emulate the features of atomicity-violation faults, we adopt the following fault model: *the access to a shared variable by 2 threads is non-atomic due to missing lock operations, where at least one of the accesses is a write to the variable.* This kind of fault is more usually referred to as *race condition.*

5.4.2 Overview of the concurrency injection technique

In order to emulate the considered fault model, we propose a SFI technique consisting of two phases. In the first phase, namely *fault injection phase*, (Subsection 5.4.3), we identify *critical sections* in which same shared variable is accessed. A critical section is a piece of code that atomically accesses shared memory variables, which begins with a lock acquisition and ends with a lock release. The fault model requires the removal of lock operations before and after a critical section.

Since concurrency faults (and Mandelbugs in general) have a low probability to be activated by random scheduling, the proposed technique includes a second phase, namely *trigger injection phase* (Subsection 5.4.4), for activating the injected fault. A fault is triggered when a thread that is executing the critical section is *preempted* and another thread that overwrites the same shared variable is scheduled. Since the shared variable accessed by a critical section is not intended to be accessed by other threads, this thread schedule corrupts the software state. Trigger injection controls thread scheduling during an experiment, in order to cause an interference between threads.

5.4.3 Fault injection phase

In the fault injection phase, information about memory accesses and lock operations is collected and analyzed to identify critical sections in which concurrency faults can be injected. This information is obtained by monitoring the execution of the system in fault-free runs. During execution, we profile shared memory accesses and lock usage (Figure 5.6). Memory and lock profiling is performed using the Valgrind profiling tool [150, 149]. This profiling is performed by submitting only one input message for each run, in order to identify which memory accesses are made under each individual input in a given set I of inputs. The inputs in the set I will be used to trigger a concurrency fault during the trigger injection phase.

More specifically, the following information is collected:

- The set of critical sections accessed for each input i;

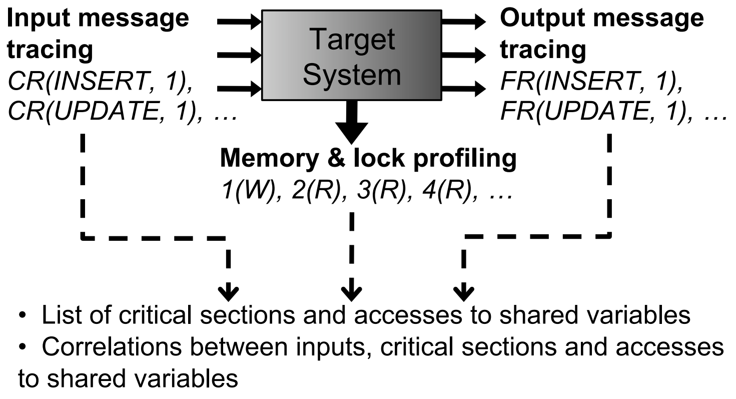

Input message tracing
CR(INSERT, 1), CR(UPDATE, 1), ...
Target System
Output message tracing
FR(INSERT, 1), FR(UPDATE, 1), ...
Memory & lock profiling
1(W), 2(R), 3(R), 4(R), ...
• List of critical sections and accesses to shared variables
• Correlations between inputs, critical sections and accesses to shared variables

Figure 5.6: Memory and lock profiling.

- The locks acquired before each critical section j;
- The set of memory accesses to shared variables made in each critical section j;
- The shared variable sv_a accessed by the access a in the critical section j;
- The type (read or write) of each access a, that is, $\text{type}(a) = R$ or $\text{type}(a) = W$.

Table 5.2 shows the shared variables for the case study, by using their symbolic name in the source code (although no information about the source code is required by the proposed fault injection technique). Moreover, 15 critical sections are identified. For each shared variable, the table shows the critical section accessing to that variable, and the type of access (e.g., if critical section 11 both reads and writes a shared variable, we write "11-RW"). The *locks* variable is an array representing the FDPs table. Each element stores a set of attributes related to an FDP request (e.g., the *callback* attribute is an identifier of the client making the request). The shared variable *state* is used by the FT Service for checkpointing;

m_state_changed is a boolean flag which is set when *state* is updated.

Table 5.2: Shared variables and critical sections in the FDPS.

Shared variable	**Critical sections and access type**
locks	1-RW 2-R 3-R 4-R 5-R 10-R 11-R 12-R 13-R 15-R
locks[i].id	1-W 2-R 11-R 15-R
locks[i].flag	1-W 2-RW 3-RW 11-RW 12-W 15-W
locks[i].callback	1-W 4-W 10-R 15-W
locks[i].mutex	1-W 2-R 3-R 11-R 12-R 15-W
locks[i].cond	1-W 2-R 3-R 11-R 12-R 15-W
state	5-W 8-R 11-R 12-R 13-W
m_state_changed	6-W 7-R 9-W 12-W 14-W

Subsequently, data on locks and memory accesses are used to identify faults to be injected. A fault injection experiment consists of the following steps:

1. Select any pair of critical sections such that:
   (a) They contain an access (respectively, a' and a'') to the same shared variable, that is, $sv_{a'} = sv_{a''}$;
   (b) The accesses a' and a'' are conflicting, that is type(a') $= W$ and type(a'') $= R$;
   (c) The same lock l is acquired by the critical sections.
2. Remove the acquisition of lock l before the critical section, and its release after the critical section;
3. Run the test using the trigger injection technique.

Removal of lock operations can be made statically, i.e., by modifying the binary or source code. Another option is to make lock operations ineffective at run-time, by wrapping functions for lock acquisition and release. Wrappers can be programmed to skip a lock operation when it is made from a particular code location (i.e., before entering and after leaving the critical section). This technique can be easily ported to several hardware/software platforms, and it introduces a negligible overhead [137]. Moreover, as discussed in the following, this option is convenient for controlling fault triggering, and it is therefore used in the proposed technique.

5.4.4 Trigger injection phase

The trigger injection phase controls thread scheduling in order to activate an injected fault. Moreover, since we aim to achieve state coverage, the trigger injection phase provides a procedure to identify the inputs that bring the system in a desired target state when the fault is activated.

Figure 5.7 shows an example of inputs (provided by the proposed procedure) that trigger a fault in the target state *2:1:0*. In order to trigger a fault in that state, a CRQ UPDATE input should be submitted when the system is in state *1:1:0*. In answer to this input, the Façade executes the critical section with the memory access a''. In order to enforce a specific thread scheduling to trigger the fault, we block the thread that is going to make the a'' access (by means of a breakpoint). Subsequently, a CR DELETE input is sent after that the system reached state *2:1:0*. The system then performs the memory access a' (which conflicts with a''); this event is intercepted by means of another breakpoint, in order to unblock the thread that is going to make the access a''. From this moment, the fault is active: in this example, threads interfere with each other because a shared variable is overwritten by thread 2 before being read by thread 1. The inputs are properly selected such that the system moves to state *2:1:0* when the conflicting accesses occur.

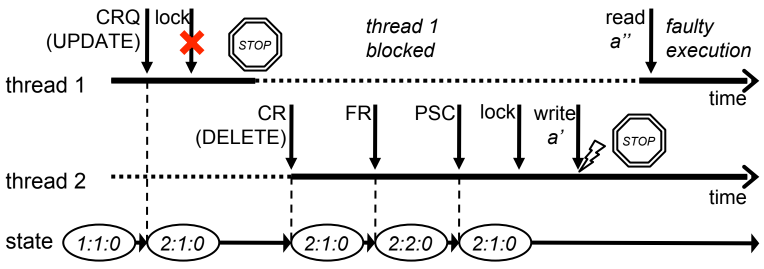


Figure 5.7: Input timing that triggers a fault in state *2:1:0*.

To avoid fault activation before the target state is reached, lock operations are made ineffective at run-time by means of wrappers (Section 5.4.3). In particular, wrappers enable lock operations when fault triggering should be avoided, and disable them when the fault triggering procedure begins. It should be noted that this solution does not affect fault representativeness, as this scenario is equivalent to a dormant fault, i.e., thread scheduling never preempts a thread when it executes a non-atomic critical section. In this way, it is possible to trigger a fault in a specific state.

Input selection. To identify inputs for triggering a fault in a target state, we design a procedure based on the FSM model of the system (described in Subsection 5.3.1). The procedure is based on the relationship between input messages and memory accesses, which is obtained during the fault injection phase (Subsection 5.4.3). This information is needed to identify which inputs the tester should submit in order to a reach a specific target state when the fault is triggered.

Table 5.3: Inputs, message sequences and critical sections in the FDPS.

Input	**Messages and Memory Accesses**		
CR/CRQ INSERT	CR/CRQ 1, 2, 3, 4, 5, 6	FR/FRQ 7, 8, 9	PSC/PSCQ 10, 11, 12, 13, 14, 7, 8, 9
CR/CRQ DELETE	CR/CRQ 2, 3, 4, 5, 6	FR/FRQ 7, 8, 9	PSC/PSCQ 10, 11, 12, 13, 14, 15, 7, 8, 9
CR/CRQ UPDATE	CR/CRQ 2, 3, 4, 5, 6	FR/FRQ 7, 8, 9	PSC/PSCQ 10, 11, 12, 13, 14, 7, 8, 9

Table 5.3 shows the relationship between inputs, message sequences (see Table 5.1), and critical sections occurring after each input. A CR/CRQ message is followed by a FR/FRQ message, which in turn is followed by a PSC/PSCQ message. However, depending on the type of request (INSERT, DELETE, or UPDATE), a different sequence of critical sections is executed. Executions of the same critical section are represented by the same integer number (e.g., critical section 2 is executed under every input, while critical section 1 is executed only under an INSERT input). Given a pair of conflicting accesses a' and a'', the proposed procedure performs the following steps:

1. Find the message sequence in Table 5.3 that causes the memory access a';
2. For this message sequence, find a sub-path (*first backward path*, *FBP*) in the FSM that ends in the target state, and that contains the message sequence (recall that edges in the FSM is associated to messages exchanged within the system); let S be the first state of this sub-path;
3. Find the message sequence in Table 5.3 that makes the memory access a'';
4. For this message sequence, find a sub-path (*second backward path*, *SBP*) in the FSM that ends in the state S, and that contains the message sequence;
5. The final path is obtained by connecting the *SBP* to the *FBP* in the state S.

The sequences of messages and critical sections may also depend on the current state of the system. This issue can be solved during the preliminary analysis by repeating, under every state of the system, the analysis of sequences caused by each input. Using the additional information about states, the algorithm can be extended to cope with state dependence (by exploring only those paths allowed under a given state). However, the sequences in the FDPS do not exhibit any state dependence, and this extension was left as future work.

Fault triggering. The inputs to be submitted by the tester for fault triggering are the initial messages of the *SBP* and the *FBP*, respectively. More specifically, the tester has to:

1. Send the first message associated to the *SBP*;

2. Wait until the thread is blocked by a breakpoint before the access a'';
3. Send the first message associated to the *FBP*;
4. Wait until the memory access a' (detected using a breakpoint); subsequently, both threads are unblocked.

Breakpoint setup. Breakpoints are exploited to drive thread scheduling, in order to cause a faulty interleaving between memory accesses. The first breakpoint is inserted before the memory access a'', in order to stop the execution of a thread until another thread makes the memory access a'. The second breakpoint is inserted after a', in order to re-enable the thread interrupted by the previous breakpoint. It should be noted that the intrusiveness due to breakpoints is negligible, since their delay (less than 1 ms in modern CPUs) is much littler than other delays in complex systems (e.g., communication and processing delays).

An example of concurrency fault trigger. We describe an example of fault trigger identified by the proposed procedure. The fault triggering path is shown in Figure 5.8, which contains a subset of the FSM in Figure 5.3. Let suppose to inject a fault in the state *2:1:0* between the critical section 8 that follows a PSC DELETE message, and critical section 2 that follows a CRQ UPDATE message (as in Figure 5.7). The two critical sections make respectively a write (a') and a read (a'') memory access to the same shared variable. First, the procedure finds a sub-path (*FBP*) ending with a PSC transition in *2:1:0* (steps 1, 2).

The sub-path should start with a CR or a CRQ transition, since all message sequences in Table 5.3 start with a CR or a CRQ request. In the example, a suitable *FBP* is: *2:1:0* → CR → *2:1:0* → FR → *2:2:0* → PSC → *2:1:0*. Subsequently, the procedure finds a second sub-path (*SBP*) ending with a CRQ transition in S = *2:1:0*, and starting with a CR or CRQ transition (steps 3, 4). In the example, the *SBP* is: *1:1:0* → CRQ → *2:1:0*. The final path is obtained by connecting the two sub-paths in reverse order (step 5): *1:1:0* → CRQ → *2:1:0* → CR → *2:1:0* → FR → *2:2:0* → PSC → *2:1:0*.

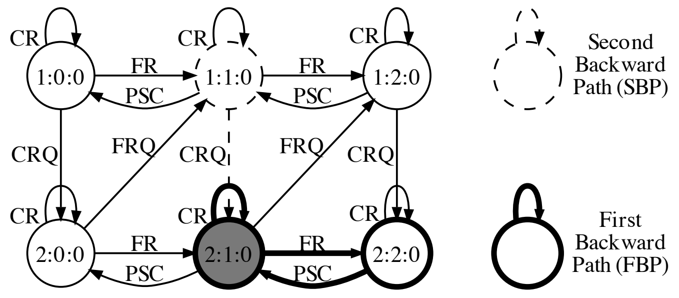


Figure 5.8: An example of path in the FSM that can trigger a fault in *2:1:0*.

To trigger the fault using this path (Figure 5.8), the tester has to send to the system a CRQ UPDATE and a CR DELETE request, in accordance to the timing shown in Figure 5.7. The system should first reach the initial state of the path (*1:1:0* in our example). The choice of the workload used to reach the initial state does not affect the experiment; in our case study, we use the workloads described in Section 5.3. The sequence of inputs and messages in Figure 5.7 are the same of the *SBP* and *FBP*, respectively.

There can be several paths that can be used to trigger a fault in a given state. However, it is better to choose the shortest path. In fact, it is possible that blocking a thread may cause the stall of the program, even if lock operations are made ineffective. For example, the thread may have acquired a logical resource by means of a boolean flag, which prevents further execution of other threads. By choosing the shortest path, such a scenario will be less likely. Nevertheless, it is neither possible to avoid this issue in all cases, nor to know *a priori* if it will happen. Therefore, a timeout has to be enforced when it is not possible to cause a specific thread scheduling [147, 158]. If memory accesses are not made within the timeout, the experiment is aborted.

5.5 Evaluation of concurrency fault injection

The concurrency fault injection technique has been used to evaluate the FDPS in those states not adequately tested during the first campaign (i.e., states in which the fault activation distribution is null or very low). To this aim, we applied the triggering technique (Section 5.4.4) in those states. From the experimental campaign, we identified four pairs of critical sections that are able to cause a failure in at least one state (Table 5.4). Figure 5.9 shows a detailed view of states in which one or more conflicting critical sections caused a failure. During the experiments, all failures occur in the same state in which a fault is activated, and the failures manifest through invalid pointer dereference.

From the analysis of results, we conclude that:

Table 5.4: Critical section pairs leading to a failure.

Shared variable	**Critical sections**	**Messages**
locks[i].cond	2-R, 1-W	CRQ, CR
locks[i].cond	11-R, 15-W	PSC, PSC
locks[i].mutex	2-R, 15-W	CR, PSC
locks[i].mutex	11-R, 15-W	PSC, PSC

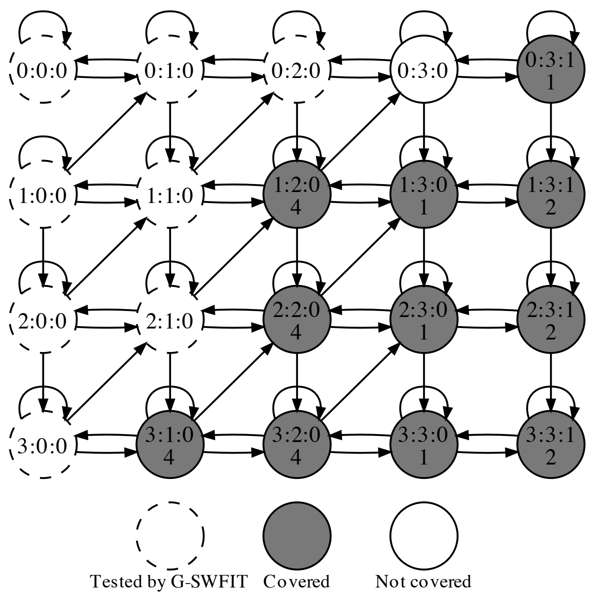


Figure 5.9: Fault activation distribution of the concurrency fault injection campaign.

- There are some conflicts that can be activated only in specific states (e.g., in *2:3:0*, only 1 out of 4 failure-prone conflicts is activated). This is due to relationship that exist between the system state and fault activation: it may not be possible to find a fault activation path in those target states for some conflicts.

- By proper fault triggering, it is possible to trigger a fault in almost all states not adequately tested by G-SWFIT. Even if these states are actually failure-prone, faults injected by G-SWFIT are seldom activated in them. Instead, the proposed technique allows to cover these states.

- Only the state *0:3:0* is not covered by both experimental campaigns. It is still possible that the system can be affected by software faults in state *0:3:0* (e.g., by different kinds of Mandelbugs). However, the increased state coverage (95%) brings a significant improvement on the confidence of the overall SFI campaign.

5.6 Summary

In this chapter, we analyzed the problem of emulating Mandelbugs, in the context of a real-world fault-tolerant system for Air Traffic Control (ATC). The state-of-the-art SFI technique G-SWFIT was evaluated. The analysis pointed out that faults injected by G-SWFIT are not representative of Mandelbugs, because most of them manifest in the early phase of the execution, and they deterministically affect both replicas. This has negative effects on the state coverage, because not covered states (35%) can potentially be affected by software faults, as demonstrated by the second SFI campaign. The technique for concurrency fault injection was able to inject and trigger faults in most of the states, reducing non-covered states down to 5%, and thus improving the confidence in the assessment of fault tolerance. Future work will encompass a broader analysis of G-SWFIT on different systems, to confirm that SFI campaigns suffer the same limitations of the ones observed in this work, due to lack of control on fault activation by G-SWFIT. Moreover, the technique could be easily extended to include deadlocks, which can be emulated by the incorrect acquisition of a set of locks in a similar way to race conditions.

Chapter 6

Conclusions and future work

This dissertation investigated the problem of fault representativeness in Software Fault Injection. This property is required in order to obtain confident conclusions about the effectiveness of fault tolerance and the safety of the overall system, by emulating the faulty conditions that the system will experience during the operational phase. In particular, this thesis addressed the problem of fault representativeness with respect to three aspects: (i) improving the representativeness of the fault model, by taking into account the location in which faults should be injected; (ii) improving the accuracy of techniques and tools that inject faults in the binary code of the target; (iii) extending Software Fault Injection to the relevant class of concurrency faults.

Along the direction of improving the representativeness of the fault model, this thesis analyzed the following important aspect: do the injected faults represent the residual faults that *escape the development process*, which actually affect the system? If this property is not achieved, the results from Software Fault Injection would be misleading, as they would not reflect the real software fault tolerance of the system. This property is determined by two factors, namely *what type of fault is injected*, and *where the fault is injected.* Past work mainly focused on the first factor, and proposed a set of realistic *fault types.* This thesis focused on the problem of *fault locations*: these should be selected based on the testing activities that are performed, and on the complexity and the relationships of software components. Experimental results from three real-world complex systems found that fault locations actually affect fault representativeness, since a significant part of faults injected using G-SWFIT are consistently detected by test cases (these faults would be easily detected and removed by developers, and thus cannot be considered as representative of residual faults). The proposed approach improves fault representativeness by selecting the most suitable components (files or functions) in which to inject faults, by using software complexity metrics and classification algorithms. The approach was effective at improving the representativeness of faultloads and reducing the faultload size (i.e., the number of fault injection experiments) at the same time.

The accuracy of Software Fault Injection in binary code is another relevant issue towards its application in real-world scenarios, such as systems based on Commercial Off-The-Shelf software, for which the source code is usually not available. However, injecting faults in binary code requires that programming constructs are recognized by looking only at the binary code, which is a difficult and error-prone task. Therefore, this thesis proposed a method for assessing the accuracy of binary-level fault injection, in order to provide confidence that faults can be accurately be injected, and to debug and to improve techniques and tools for injecting binary-level faults. The method was adopted for assessing G-SWFIT and an industrial tool based on this technique, by evaluating the accuracy of faults injected in a real-world complex software system. The analysis found several pitfalls that can occur in the implementation of G-SWFIT, that need to be avoided in order to achieve an acceptable degree of accuracy. The results highlight that it is difficult to develop an accurate binary-level fault injection tool. Therefore, significant efforts could be required to develop a tool tailored to a new CPU architecture or compiler. This problem could be mitigated by a systematic approach for analyzing how software faults are translated in binary code for a given CPU and compiler, in a similar way to the proposed method. In future work, the proposed method could be extended to support the development of new fault injection tools, in order to reduce development efforts.

Finally, this thesis investigated the problem of emulating concurrency faults. It is unfeasible to remove these software faults during development, since they manifest themselves depending on complex interactions with the environment (e.g., thread scheduling) and with the state of the system. In fact, these faults represent a significant part of faults affecting complex software. Moreover, fault tolerance mechanisms are often specifically tailored for this kind of faults. Therefore, it becomes important to assess a system through the injection of concurrency faults, which instead have been overlooked by existing techniques. This thesis proposed a new technique for emulating these faults: it injects both *faults in the executable code* and the *triggering conditions that make these faults manifest themselves*, by controlling locking operations and thread scheduling. An experiment in a fault-tolerant system for Air Traffic Control confirms that this is a promising approach towards the emulation of concurrency faults, since the proposed technique more closely emulates the behavior of concurrency faults than traditional techniques such as G-SWFIT. In future work, the technique could include both race conditions and deadlocks, which represent the vast majority of concurrency faults, and the effectiveness and scalability of this technique could be evaluated in other complex software systems.

Bibliography

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990.

[2] IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-1993*, 1994.

[3] IEEE Standard for Information Technology–Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) Amendment 1: Realtime Extension [C Language]. *IEEE Std 1003.1b-1993*, 1994.

[4] NASA Software Safety Guidebook. *NASA-GB-8719.13*, 2004.

[5] IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, 2010.

[6] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic Object-Oriented Fault Injection Tool. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 83–88, 2001.

[7] A. Albinet, J. Arlat, and J.C. Fabre. Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 867–876. IEEE Computer Society, 2004.

[8] A.M. Ambrosio, F. Mattiello-Francisco, VA Santiago, W.P. Silva, and E. Martins. Designing Fault Injection Experiments Using State-Based Model to Test a Space Software. *Lecture Notes in Computer Science*, 4746, 2007.

[9] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. Intl. Conf. on Software Engineering*, pages 402–411. ACM, 2005.

[10] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, and D. Powell. Fault Injection for Dependability Validation: A Methodology and Some Applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.

[11] J. Arlat, A. Costes, Y. Crouzet, J.C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, 1993.

[12] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G.H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, pages 1115–1133, 2003.

[13] J. Arlat, J.C. Fabre, M. Rodríguez, and F. Salles. Dependability of COTS Microkernel-Based Systems. *IEEE Transactions on Computers*, pages 138–163, 2002.

[14] T.F. Arnold. The Concept of Coverage and Its Effect on the Reliability Model of Repairable Systems. *IEEE Transactions on Computers*, 22(6):251–254, June 1973.

[15] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, 1985.

[16] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[17] D. Avresky, J. Arlat, J.C. Laprie, and Y. Crouzet. Fault Injection for Formal Testing of Fault Tolerance. *IEEE Transactions on Reliability*, 45(3):443–455, 1996.

[18] Meera Balakrishnan, Antonio Puliafito, Kishor S. Trivedi, and Yannis Viniotis. Buffer losses vs. deadline violations for ABR traffic in an ATM switch: A computational approach. *Telecommunication Systems*, 7(1-3):105–123, 1997.

[19] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, 1990.

[20] V.R. Basili and B.T. Perricone. Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1):42–52, 1984.

[21] T. Basso, R. Moraes, B.P. Sanches, and M. 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*, pages 1–13, 2009.

[22] Fabrice Bellard. *QEMU virtualization software*. `http://qemu.org`, year=2011,.

[23] B.W. Boehm. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, Jan. 1984.

[24] A. Bondavalli, A. Ceccarelli, L. Falai, and M. Vadursi. Foundations of Measurement Theory Applied to the Evaluation of Dependability Attributes. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 522–533. IEEE, 2007.

[25] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano. Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications. *IEEE Transactions on Dependable and Secure Computing*, 1(4):223–237, 2004.

[26] W.G. Bouricius, W.C. Carter, and P.R. Schneider. Reliability Modeling Techniques for Self-Repairing Computer Systems. In *Proc. 24th Nat'l Conf. ACM*, pages 295–309, 1969.

[27] P. Broadwell, N. Sastry, and J. Traupman. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and self-MANaged Systems*, 2002.

[28] J. Carreira, H. Madeira, and J.G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.

[29] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: Software Fault Injection and Monitoring in Processor Functional Units. In *Proc. Intl. Conf. on Dependable Computing for Critical Applications*, pages 135–149, 1995.

[30] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo. Memory Leak Analysis in Mission-Critical Middleware. *Journal of Systems and Software*, 83(9), 2010.

[31] J.K. Chaar, M.J. Halliday, I.S. Bhandari, and R. Chillarege. In-Process Evaluation for Software Inspection and Test. *Software Engineering, IEEE Transactions on*, 19(11):1055–1070, 1993.

[32] R. Chandra, R.M. Lefever, K.R. Joshi, M. Cukier, and W.H. Sanders. A Global-State-Triggered Fault Injector for Distributed System Evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605, 2004.

[33] S. Chandra and PM Chen. How Fail-Stop are Faulty Programs? In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 240–249, 1998.

[34] S. Chandra and P.M. Chen. Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 97–106. IEEE, 2000.

[35] R. Chillarege. Understanding Bohr-Mandel bugs through ODC Triggers and a case study with empirical estimations of their field proportion. In *IEEE Third International Workshop on Software Aging and Rejuvenation, IEEE Intl. Symp. on Software Reliability Engineering.*

[36] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.Y. Wong. Orthogonal Defect Classification–A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

[37] R. Chillarege and N.S. Bowen. Understanding large system failures-a fault injection experiment. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 356–363. IEEE, 1988.

[38] R. Chillarege, W.L. Kao, and R.G. Condit. Defect Type and its Impact on the Growth Curve. In *Proc. 13th Intl. Conf. on Software Engineering*, pages 246–255, 1991.

[39] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. ACM Symp. on Operating Systems Principles.* ACM, 2001.

[40] M.B. Chrissis, M. Konrad, and S. Shrum. *CMMI: Guidelines for process integration and product improvement.* Addison-Wesley Professional, 2003.

[41] J. Christmansson and R. Chillarege. Generation of an Error Set that Emulates Software Faults based on Field Data. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 304–313, 1996.

[42] J. Christmansson, M. Hiller, and M. Rimen. An Experimental Comparison of Fault and Error Injection. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 369–378, 1998.

[43] J. Christmansson and P. Santhanam. Error Injection Aimed at Fault Removal in Fault Tolerance Mechanisms–Criteria for Error Selection using Field Data on Software Faults. In *Proc. of Intl. Symp. on Software Reliability Engineering*, pages 175–184, 1996.

[44] ObjectWeb Consortium. *CARDAMOM—An Enterprise Middleware for Building Mission and Safety Critical Applications.* `http://cardamom.ow2.org/`, 2011.

[45] Oracle Corporation. *MySQL Market Share.* `http://www.mysql.com/why-mysql/marketshare/`, 2011.

[46] Standard Performance Evaluation Corporation. *SPECweb99 v1.02.* `http://www.spec.org/web99/`, 2009.

[47] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo. Software Aging Analysis of the Linux Operating System. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, pages 71–80. IEEE, 2010.

[48] Transaction Processing Performance Council. *TPC Benchmark C (TPC-C) v5.11.* `http://www.tpc.org/tpcc/`, 2010.

[49] F. Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, C-31(6):531–540, 1982.

[50] M. Cukier, D. Powell, and J. Ariat. Coverage estimation methods for stratified fault-injection. *IEEE Transactions on Computers*, 48(7):707–723, 1999.

[51] S. Dawson, F. Jahanian, and T. Mitton. Experiments on six commercial TCP implementations using a software fault injection tool. *Software Practice and Experience*, 27(12):1385–1410, 1997.

[52] S. Dawson, F. Jahanian, T. Mitton, and T.L. Tung. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, 1996.

[53] M.E. Delamaro and J.C. Maldonado. Proteum—A Tool for the Assessment of Test Adequacy for C Programs. In *Proc. Conf. Performability in Computer Systems*, pages 79–95, 1996.

[54] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[55] J. DeVale, P. Koopman, and D. Guttendorf. The Ballista software robustness testing service. In *Testing Computer Software Conference*, 1999.

[56] D. Dig, J. Marrero, and M.D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *Proc. Intl. Conf. on Software Engineering*, pages 397–407, 2009.

[57] CP Dingman and J. Marshall. Measuring Robustness of a Fault-Tolerant Aerospace System. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 522–527. IEEE Computer Society, 1995.

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

[59] J. Durães and H. Madeira. Characterization of Operating Systems Behavior in the Presence of Faulty Drivers through Software Fault Emulation. In *Proc. IEEE Pacific Rim Intl. Symp. on Dependable Computing*, pages 201–209. IEEE Computer Society, 2002.

[60] J. Duraes and H. Madeira. Emulation of Software Faults by Educated Mutations at Machine-Code Level. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 329–340, 2002.

[61] J. Duraes and H. Madeira. Definition of Software Fault Emulation Operators: A Field Data Study. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 105–114, 2003.

[62] J. Durães and H. Madeira. Generic Faultloads Based on Software Faults for Dependability Benchmarking. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, 2004.

[63] J. Durães, M. Vieira, and H. Madeira. Multidimensional Characterization of the Impact of Faulty Drivers on the Operating Systems Behavior. *IEICE Transactions on Information and Systems*, 86(12):2563–2570, 2003.

[64] J. Durães, M. Vieira, and H. Madeira. Dependability benchmarking of web-servers. In *Computer safety, reliability, and security: 23rd international conference, SAFECOMP 2004, Potsdam, Germany, September 21-24, 2004: proceedings*, volume 3219, pages 297–310. Springer-Verlag New York Inc, 2004.

[65] J.A. Durães and H.S. Madeira. Emulation of Software faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[66] D.L. Dvorak. *NASA Study on Flight Software Complexity.* NASA Office of Chief Engineer, 2009.

[67] S.G. Eick, C.R. Loader, M.D. Long, L.G. Votta, and S. Vander Wiel. Estimating Software Fault Content Before Coding. In *Proc. Intl. Conf. on Software Engineering*, pages 59–65. ACM, 1992.

[68] EnterpriseDB. *EnterpriseDB's Postgres Plus users by Application Type.* `http://www.enterprisedb.com/customer-success/customers-by-application-workload`, 2011.

[69] European Cooperation for Space Standardization. ECSS-E-70-41A – Ground Systems and Operations: Telemetry and Telecommand Packet Utilization, 2003.

[70] N.E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[71] N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., 1998.

[72] C. Fetzer, P. Felber, and K. Högstedt. Automatic Detection and Masking of Nonatomic Exception Handling. *IEEE Transactions on Software Engineering*, 30:547–560, 2004.

[73] C. Fetzer and Z. Xiao. HEALERS: a toolkit for enhancing the robustness and security of existing applications. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 317–322. IEEE.

[74] C. Fetzer and Z. Xiao. An automated approach to increasing the robustness of C libraries. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 155–164. IEEE, 2002.

[75] J. Fonseca and M. Vieira. Mapping software faults with web security vulnerabilities. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 257–266. Ieee, 2008.

[76] J. Fonseca, M. Vieira, and H. Madeira. Training Security Assurance Teams Using Vulnerability Injection. In *Proc. IEEE Pacific Rim Intl. Symp. on Dependable Computing*, pages 297–304, 2008.

[77] J. Fonseca, M. Vieira, and H. Madeira. Vulnerability & Attack Injection for Web Applications. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 93–102. IEEE, 2009.

[78] International Organization for Standardization. Product development: software level. *ISO/DIS 26262-6*, 2009.

[79] U.S.-Canada Power System Outage Task Force. *Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations.* U.S. Energy Department, 2004.

[80] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows xp kernel crash analysis. In *Proc. USENIX Large Installation System Administration Conf.*, pages 101–111, 2006.

[81] A.K. Ghosh and M. Schmid. An approach to testing COTS software for robustness to operating system exceptions and errors. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, pages 166–174, 1999.

[82] A.K. Ghosh, M. Schmid, and V. Shah. Testing the Robustness of Windows NT Software. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 231–235, 1998.

[83] J. Gray. Why Do Computers Stop and What Can Be Done About It? In *Proc. Symp. on Reliability in Distributed Software and Database Systems*, pages 3–11, 1985.

[84] J. Gray. A Census of Tandem System Availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, 1990.

[85] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi. Analysis of Software Aging in a Web Server. *IEEE Transactions on Reliability*, 55(3):480–491, 2006.

[86] M. Grottke, A.P. Nikora, and K.S. Trivedi. An empirical investigation of fault types in space mission system software. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 447–456. IEEE, 2010.

[87] M. Grottke and K.S. Trivedi. Software Faults, Software Aging and Software Rejuvenation. *Journal of the Reliability Engineering Association of Japan*, 27(7):425–438, 2005.

[88] M. Grottke and K.S. Trivedi. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. *IEEE Computer*, 40(2):107–109, 2007.

[89] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 340–347. IEEE, 1989.

[90] R.G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, 1977.

[91] S. Han, KG Shin, and HA Rosenberg. DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment. In *Proc. Intl. Computer Performance and Dependability Symp.*, pages 204–213, 1995.

[92] S. Henry and D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, pages 510–518, 1981.

[93] M. Hiller, A. Jhumka, and N. Suri. An approach for analysing the propagation of data errors in software. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 161–170, 2001.

[94] M. Hiller, A. Jhumka, and N. Suri. EPIC: Profiling the propagation and effect of data errors in software. *Computers, IEEE Transactions on*, 53(5):512–530, 2004.

[95] M.C. Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[96] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, 1995.

[97] JJ Hudak, B.H. Suh, DP Siewiorek, and Z. Segall. Evaluation and Comparison of Fault-Tolerant Software Techniques. *IEEE Transactions on Reliability*, 42(2):190–204, 1993.

[98] Object Computing Inc. *The ACE ORB*. `http://www.theaceorb.com/`, 2011.

[99] Real-Time Innovations. *RTI Data Distribution Service*. `http://www.rti.com/`, 2011.

[100] I. Irrera, J. Durães, M. Vieira, and H. Madeira. Towards Identifying the Best Variables for Failure Prediction Using Injection of Realistic Software Faults. In *Proc. IEEE Pacific Rim Intl. Symp. on Dependable Computing*, pages 3–10. IEEE, 2010.

[101] Dependability Benchmarking Project (IST-2000-25425). DBench final report. 2004.

[102] M. Coutinho S. Santos J. Rufino, S. Filipe and J. Windsor. ARINC 653 Interface in RTEMS. In *Data Systems in Aerospace Conf.*, 2007.

[103] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. Experimental Analysis of the Errors Induced into Linux by Three Fault Injection Techniques. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 331–336. IEEE Computer Society, 2002.

[104] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, and T. Marteau. Analysis of the Effects of Real and Injected Software Faults: Linux as a Case Study. In *Proc. IEEE Pacific Rim Intl. Symp. on Dependable Computing*, pages 51–58. IEEE Computer Society, 2002.

[105] T. Jarboui, J. Arlat, Y. Crouzet, K. Kanoun, and T. Marteau. Impact of internal and external software faults on the linux kernel. *IEICE Transactions on Information and Systems*, 86(12):2571–2578, 2003.

[106] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, pages 649–678, 2001.

[107] A. Jin and J. Jiang. Fault Injection Scheme for Embedded Systems at Machine Code Level and Verification. In *Proc. IEEE Pacific Rim Intl. Symp. on Dependable Computing*, pages 55–62. IEEE, 2009.

[108] A. Johansson and N. Suri. Error propagation profiling of operating systems. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 86–95. IEEE, 2005.

[109] A. Johansson, N. Suri, and B. Murphy. On the impact of injection triggers for OS robustness evaluation. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, pages 127–126. IEEE, 2007.

[110] A. Johansson, N. Suri, and B. Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 502–511. IEEE, 2007.

[111] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 14–19. Citeseer, 1987.

[112] A. Kalakech, K. Kanoun, Y. Crouzet, and J. Arlat. Benchmarking The Dependability of Windows NT4, 2000 and XP. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 681–686. IEEE Computer Society, 2004.

[113] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. Ferrari: A tool for the validation of system dependability properties. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 336–344. IEEE, 1992.

[114] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. Emax - an automatic extractor of high-level error models. In *AIAA Computing in Aerospace Conference*, pages 1297–1306, 1993.

[115] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transactions on Computers*, 44(2):248–260, 1995.

[116] K. Kanoun, Y. Crouzet, A. Kalakech, A.E. Rugina, and P. Rumeau. Benchmarking the Dependability of Windows and Linux Using PostMark Workloads. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 11–20. IEEE Computer Society, 2005.

[117] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems.* Wiley-IEEE Computer Society, 2008.

[118] W.-I. Kao and R.K. Iyer. DEFINE: A Distributed Fault Injection and Monitoring Environment. In *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259, 1994.

[119] W.-I. Kao, R.K. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior under Faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, 1993.

[120] J. Katcher. Postmark: A New File System Benchmark. Network Appliance Technical Report TR-3022, 1997.

[121] K.N. King and A.J. Offutt. A Fortran Language System for Mutation-based Software Testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[122] J.C. Knight. Safety critical systems: challenges and directions. In *Proc. Intl. Conf. on Software Engineering*, pages 547–550. IEEE, 2002.

[123] P. Koopman and J. DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 30–37. IEEE Computer Society, 1999.

[124] P. Koopman and J. DeVale. The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE Transactions on Software Engineering*, 26(9):837–848, 2000.

[125] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 230–239. IEEE, 1998.

[126] I. Lee and RK Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 20–29, 1993.

[127] N.G. Leveson. Role of software in spacecraft accidents. *Journal of Spacecraft and Rockets*, 41(4):564–575, 2004.

[128] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. In *Proc. 1st workshop on Architectural and System Support for Improving Software Dependability*, pages 25–33, 2006.

[129] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: a review. *ACM Computing Surveys (CSUR)*, 33(2):177–208, 2001.

[130] B. Littlewood and L. Strigini. Software Reliability and Dependability: A Roadmap. In *Proc. Conf. on the Future of Software Engineering*, pages 175–188. ACM, 2000.

[131] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proc. Intl. Conf. on Architecture Support for Programming Languages and Operating Systems*, 2008.

[132] M.R. Lyu. *Software Fault Tolerance*. John Wiley & Sons, 1995.

[133] H. Madeira, D. Costa, and M. Vieira. On the Emulation of Software Faults by Software Fault Injection. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 417–426, 2000.

[134] P. Maes. Concepts and experiments in computational reflection. In *Proc. Conf. on Object-Oriented Programming Systems, Languages and Applications*. ACM, 1987.

[135] S. Maffeis and DC Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications Magazine*, 35(2):56–60, 1997.

[136] P.D. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *Proc. USENIX Annual Technical Conf.*, pages 23–23. USENIX Association, 2010.

[137] P.D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 379–388. IEEE, 2009.

[138] E. Martins, C.M.F. Rubira, and N.G.M. Leme. Jaca: A Reflective Fault Injection Tool based on Patterns. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 483–487, 2002.

[139] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, pages 2–13, 2007.

[140] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, 1998.

[141] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[142] R. Moraes, R. Barbosa, J. Durães, N. Mendes, E. Martins, and H. Madeira. Injection of Faults at Component Interfaces and Inside the Component Code: Are They Equivalent? In *Proc. IEEE European Dependable Computing Conf.*, pages 53–64, 2006.

[143] R. Moraes, J. Durães, R. Barbosa, E. Martins, and H. Madeira. Experimental Risk Assessment and Comparison using Software Fault Injection. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 512–521, 2007.

[144] R. Moraes and E. Martins. An Architecture-based Strategy for Interface Fault Injection. In *Workshop on Architecting Dependable Systems, IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, 2004.

[145] A. Mukherjee and D.P. Siewiorek. Measuring software dependability by robustness benchmarking. *Software Engineering, IEEE Transactions on*, 23(6):366–378, 1997.

[146] J.D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, Inc., 1987.

[147] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proc. Symp. on Operating Systems Design and Implementation*, pages 267–280, 2008.

[148] Roberto Natella. *SAFE SoftwAre Fault Emulation tool.* `http://wpage.unina.it/roberto.natella/`, 2011.

[149] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proc. Intl. Conf. on Virtual Execution Environments*, pages 65–74. ACM, 2007.

[150] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.

[151] W.T. Ng, CM Aycock, G. Rajamani, and PM Chen. Comparing disk and memory's resistance to operating system crashes. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, pages 185–194. IEEE Computer Society, 1996.

[152] W.T. Ng and PM Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proc. 29th Intl. Symp. on Fault-Tolerant Computing*, pages 76–83, 1999.

[153] W.T. Ng and P.M. Chen. The design and verification of the rio file cache. *IEEE Transactions on Computers*, 50(4):322–337, 2001.

[154] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[155] A.J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proc. Intl. Conf. on Software Engineering*, pages 100–107. IEEE Computer Society Press, 1993.

[156] J. Ohlsson, M. Rimen, and U. Gunneflo. A study of the effects of transient fault injection into a 32-bit risc with built-in watchdog. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 316–325. IEEE.

[157] J. Pan, P. Koopman, D.P. Siewiorek, Y. Huang, R. Gruber, and M.L. Jiang. Robustness Testing and Hardening of CORBA ORB Implementations. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 141–150. IEEE Computer Society, 2001.

[158] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[159] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. SymPLFIED: Symbolic program-level fault Injection and error detection framework. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 472–481. IEEE, 2008.

[160] K. Pattabiraman, G.P. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, 2010.

[161] P. Popov and L. Strigini. Assessing Asymmetric Fault-Tolerant Software. In *Proc. Intl. Symp. on Software Reliability Engineering*, pages 41–50, 2010.

[162] D. Powell. Failure Mode Assumptions and Assumption Coverage. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 386–395, 1992.

[163] D. Powell, E. Martins, J. Arlat, and Y. Crouzet. Estimators for Fault Tolerance Coverage Evaluation. *IEEE Transactions on Computers*, 44(2):261–274, 1995.

[164] GNU Project. *GCC documentation.* `http://gcc.gnu.org/onlinedocs/gcc/`, year=2011,.

[165] Kai Qian, David den Haring, and Li Cao. *Embedded Software Development with C.* Springer, 2009.

[166] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. In *Proc. ACM Symp. on Operating Systems Principles*, 2005.

[167] G.L. Ries, G.S. Choi, and R.K. Iyer. Device-level transient fault modeling. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 86–94. IEEE, 1994.

[168] M. Rodríguez, F. Salles, J.C. Fabre, and J. Arlat. MAFALDA: Microkernel assessment by fault injection and design aid. *Dependable Computing—EDCC-3*, pages 143–160, 1999.

[169] Research Triangle Institute (RTI). *The Economic Impacts of Inadequate Infrastructure for Software Testing.* National Institute of Standards and Technology (NIST), 2002.

[170] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010.

[171] F. Salfner and M. Malek. Using hidden semi-markov models for effective online failure prediction. In *Proc. IEEE Intl. Symp. on Reliable Distributed Systems*, pages 161–174. IEEE, 2007.

[172] F. Salles, M. Rodriguez, J.-C. Fabre, and J. Arlat. MetaKernels and Fault Containment Wrappers. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 22–29, 1999.

[173] B.P. Sanches, T. Basso, and R. Moraes. J-SWFIT: A Java Software Fault Injection Tool. In *Proc. Latin American Symp. on Dependable Computing*, 2011.

[174] W. Sanders and J. Meyer. Stochastic Activity Networks: Formal Definitions and Concepts. *Lectures on Formal Methods and Performance Analysis*, pages 315–343, 2001.

[175] A. Schiper, K. Birman, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991.

[176] Inc. Scientific Toolworks. *SciTools Understand.* `http://www.scitools.com`, 2011.

[177] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. Fiat-fault injection based automated testing environment. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 102–107. IEEE, 1988.

[178] C.P. Shelton, P. Koopman, and K. Devale. Robustness testing of the Microsoft Win32 API. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 261–270. IEEE, 2000.

[179] D. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman and Hall/CRC, 2004.

[180] L. Silva, H. Madeira, and JG Silva. Software Aging and Rejuvenation in a SOAP-based Server. In *Proc. IEEE Intl. Symp. on Network Computing and Applications*, pages 56–65, 2006.

[181] Critical Software. *Critical Software products and services.* `http://www.criticalsoftware.com/`, 2011.

[182] M. Sridharan and A.S. Namin. Prioritizing Mutation Operators Based on Importance Sampling. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, pages 378–387. IEEE.

[183] K. Srinivasan and D. Fisher. Machine Learning Approaches to Estimating Software Development Effort. *IEEE Transactions on Software Engineering*, pages 126–137, 1995.

[184] D.T. Stott, B. Floering, Z. Kalbarczyk, and R.K. Iyer. A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors. In *Proc. Intl. Computer Performance and Dependability Symp.*, pages 91–100, 2000.

[185] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(3):204–226, 1985.

[186] M. Subkraut and C. Fetzer. Automatically finding and patching bad error handling. In *Dependable Computing Conference, 2006. EDCC'06. Sixth European*, pages 13–22. IEEE, 2006.

[187] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability: A Study of Field Failures in Operating Systems. In *Proc. Intl. Symp. on Fault-Tolerant Computing*, pages 2–9, 1991.

[188] M. Susskraut and C. Fetzer. Robustness and security hardening of COTS software libraries. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 61–71. IEEE, 2007.

[189] A. Thakur, R.K. Iyer, L. Young, and I. Lee. Analysis of failures in the Tandem NonStop-UX operating system. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, pages 40–50. IEEE, 1995.

[190] S.K. Thompson. Sample Size for Estimating Multinomial Proportions. *The American Statistician*, 41(1):42–46, 1987.

[191] *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, pages 407–429. John Wiley & Sons, 1st edition, 2005.

[192] K.S. Trivedi. Keynote Talk: Software Aging and Rejuvenation: Modeling and Analysis. In *Workshop on Self-Healing, Adaptive and Self-MANaged Systems*, 2002.

[193] T.K. Tsai, M.C. Hsueh, H. Zhao, Z. Kalbarczyk, and R.K. Iyer. Stress-based and path-based fault injection. *Computers, IEEE Transactions on*, 48(11):1183–1201, 1999.

[194] T.K. Tsai and R.K. Iyer. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In *Proc. Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems*, 1995.

[195] K. Vaidyanathan and K.S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2):124–137, 2005.

[196] P.C. Véras, E. Villani, A.M. Ambrosio, N. Silva, M. Vieira, and H. Madeira. Errors on Space Software Requirements: A Field Study and Application Scenarios. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, pages 61–70. IEEE.

[197] M. Vieira and H. Madeira. Benchmarking the Dependability of Different OLTP Systems. In *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pages 305–310, 2003.

[198] M. Vieira and H. Madeira. A dependability benchmark for oltp application environments. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 742–753. VLDB Endowment, 2003.

[199] M. Vieira, H. Madeira, I. Irrera, and M. Malek. Fault injection for failure prediction methods validation. In *Workshop on Hot Topics in System Dependability, IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, 2009.

[200] K.P. Vo, Y.M. Wang, PE Chung, and Y. Huang. Xept: A software instrumentation method for exception handling. In *Proc. IEEE Intl. Symp. on Software Reliability Engineering*, pages 60–69. IEEE Computer Society, 1997.

[201] J. Voas, L. Morell, and K. Miller. Predicting where faults can hide from testing. *Software, IEEE*, 8(2):41–48, 1991.

[202] J.M. Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6):53–59, 1998.

[203] J.M. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly "Good" Software Can Behave. *IEEE Software*, 14(4):73–83, 1997.

[204] E.J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.

[205] S. Winter, C. Sârbu, N. Suri, and B. Murphy. The impact of fault models on software robustness evaluations. In *Proc. Intl. Conf. on Software Engineering*, pages 51–60. ACM, 2011.

[206] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[207] W.E. Wong and A.P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.