Emulation of Software Faults: A Field Data Study and a Practical Approach

João A. Durães, *Member*, *IEEE*, and Henrique S. Madeira, *Member*, *IEEE*

Abstract—The injection of faults has been widely used to evaluate fault tolerance mechanisms and to assess the impact of faults in computer systems. However, the injection of software faults is not as well understood as other classes of faults (e.g., hardware faults). In this paper, we analyze how software faults can be injected (emulated) in a source-code independent manner. We specifically address important emulation requirements such as fault representativeness and emulation accuracy. We start with the analysis of an extensive collection of real software faults. We observed that a large percentage of faults falls into well-defined classes and can be characterized in a very precise way, allowing accurate emulation of software faults through a small set of emulation operators. A new software fault injection technique (G-SWFIT) based on emulation operators derived from the field study is proposed. This technique consists of finding key programming structures at the machine code-level where high-level software faults can be emulated. The fault-emulation accuracy of this technique is shown. This work also includes a study on the key aspects that may impact the technique accuracy. The portability of the technique is also discussed and it is shown that a high degree of portability can be achieved.

Index Terms—Fault injection, software faults, software reliability.

1 Introduction

This paper addresses the emulation of software faults (i.e., program defects or bugs) for software reliability and dependability assessment purposes. Several studies show a clear predominance of software faults as the root cause of computer failures [1], [2], [3], [4] and, given the huge complexity of today's software, the weight of software faults on overall system dependability will tend to increase. The complete elimination of software defects during software development process is very difficult to attain in practice. In addition to well-known technical difficulties of the software development and testing process [5], [6], practical constraints such as the intense pressure to shrink time-to-market and cost of software contribute to the difficulties in assuring 100 percent defect-free software. Therefore, the current scenario in the computer industry is having systems in which software defects do exist but no one knows exactly where they are, when they will reveal themselves, and, above all, the possible consequences of the activation of the software faults. In a world where components-of-the-shelf (COTS) are used more and more to build larger systems, the residual software faults represent a growing risk.

Many software reliability models and measurement procedures have been proposed for the prediction and estimation of measures of quality such as the number of faults remaining in a given software package. However, it is difficult to handle the extreme complexity of today's software and the models also have to take into account aspects such as the interaction among software components in faulty situations and realistic working profile in order to give a measure of the possible impact of residual faults on the operation and on the end user.

The clear trend in the software industry toward a component-oriented development model compels the software vendors to consider the use of commonly available general-purpose components (referred to as components-of-the-shelf—COTS) more and more. This leads to software products that are in fact a collection of small components from a variety of sources. A very important question in this scenario is how to estimate the behavior of the whole system when one of its components (either a COTS or a specifically developed component) behaves in an erroneous manner due to the activation of residual software faults.

We propose the injection of software faults to help with this problem. Fault injection techniques emulating hardware faults have been extensively used in the last two decades to evaluate specific fault tolerance mechanisms and to assess the impact of faults in systems. However, although the injection of software faults was already recognized as potentially very useful [7], [8], no general approach to inject residual software faults at the executable code of the targets in an accurate manner has been proposed so far.

In addition to the major problem of knowing how to emulate software faults realistically, it is also necessary to define how to use a possible technique to inject this kind of fault, as its use is not as obvious as the classical fault injection that emulates hardware faults. The establishment of a utilization scenario for the injection of software faults from the beginning is important to help in identifying the requirements put on the accurate emulation of software faults, which is the main subject of our research.

In this context, we propose the use of software fault injection according to the following principle: Faults are

- *J.A. Durães is with the Instituto Superior de Engeharia de Coimbra, Rua Pedro Nunes—Quinta da Nora, 3030-199 Coimbra, Portugal. E-mail: judraes@isec.pt.*
- *H.S. Madeira is with the Departamento de Endenharia Informática, University of Coimbra, Polo II—Pinhal de Marrocos, 3030-290 Coimbra, Portugal. E-mail: henrique@dei.uc.pt.*

Manuscript received 5 Feb. 2005; revised 29 Mar. 2006; accepted 13 Sept. 2006; published online 6 Nov. 2006.
Recommended for acceptance by B. Littlewood.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0029-0205.

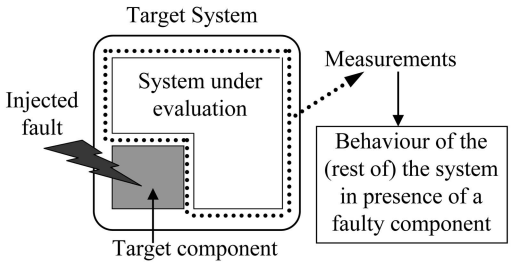

Fig. 1. Software fault injection and system observation.

injected in a given component to evaluate the behavior of the overall system in the presence of that faulty component. A fundamental aspect of this scenario is the clear separation between the target component (the one subject to the injection of faults) and the system under observation. This is required to avoid the problem of changing the system under evaluation, as the emulation of software faults will always require the introduction of small changes in the target code. More specifically, what is evaluated is the system behavior when one of its components is faulty and not the behavior of the faulty component itself (see Fig. 1). Considering the component-based approach generally used for software development, the identification of the software components normally results from the software architecture. However, the granularity used to define the software components can be determined according to the needs and a much more detailed grain can be used.

Using the scenario proposed above (Fig. 1), we identify three major uses for the emulation of software faults:

1. **Validation of fault-tolerant mechanisms**. As typical residual software faults in deployed systems are rarely triggered, the validation of fault-tolerant mechanisms can be achieved by accelerating the software fault activation through the emulation of software faults [9].
2. **Prediction of worst-case scenarios and experimental risk assessment**. The emulation of software faults can be used as a way to quantify the impact of software faults from the user's point of view and get a quantitative idea of the potential risk represented by residual faults. This allows the optimization of the testing phase effort by performing risk assessment and prediction of worst-case scenarios [8]. For example, if the injection of software faults in a given component causes a high percentage of catastrophic failures in the system, it means that residual software faults in that component may represent a high risk and more effort should be put into the testing of the component or the system design should be changed to mitigate the impact of faults in that specific component.
3. **Dependability benchmarking**. One recent research effort is centered on the definition of dependability benchmarks as a standard procedure to assess measures related to the behavior of a computer system or computer component in the presence of faults [10], [11], [12], [13], [14], [15], [16]. Techniques to emulate software faults are essential for the benchmark faultload.

It is widely accepted that existing fault injection technologies can emulate hardware faults, either transient or permanent, using simple bit-flip or stuck-at. However, the realistic emulation of residual software faults by fault injection at the target executable code is much more difficult and is still a rather obscure step. In fact, the problem of emulating software faults is intrinsically difficult. Software faults are by nature human-made faults (at the design, implementation, etc.) and it is extremely difficult to model human errors.

The key issue surrounding the injection of software faults is **fault representativeness**. Ideally, the injected faults should precisely emulate the **hidden defects of a program**, which is obviously not possible because these defects are not known in advance (if we knew the bugs, we would fix them beforehand). Given this impossibility, the correct emulation of software faults by fault injection requires, in practice, the following:

1. Identification and characterization of the most important classes of software faults and estimation of the relative percentages of these classes in real programs. The relevant faults are those that belong to representative fault types of the real residual defects found in deployed software systems.
2. Techniques to inject faults that generate errors or induce erroneous program behavior similar to the ones caused by specific classes of real software faults. That is, what is important is to avoid the injection of faults that cause errors that cannot be generated by a given class of software fault.

The identification of the most likely classes of software faults is clearly the most difficult problem. In principle, this identification could be achieved by using information about the development process, the team that developed the software, specific metrics of the code (code size, number of variables, complexity of data structures, etc), or field data about previous faults discovered in the program. However, considering that the source code of the system/component under evaluation may not available, which is the case with most COTS components, it is not likely that the adequate characterization of software faults can be obtained from knowledge on the development teams/processes or from code complexity metrics, which are difficult or impossible to obtain when the source code is not available.

Field data on real software faults is probably the most important resource to understand software faults and to help in the characterization of relevant fault attributes such as fault occurrence frequency per fault type. Unfortunately, data on real software faults is not commonly available and research works using real faults are rare [2], [4], [17], [18].

Given the problems presented above, this paper addresses the key issue of fault representativeness, and the general problem of injection of software faults, in the following way:

1. Presents the analysis of a large set of real software faults (more than 650 software faults). The source of the faults is a set of open source programs written in C including both application programs and operating

system code. The detailed analysis of these faults provides valuable insight on important fault properties, such as fault relationship with common high-level language structures and fault effects on program execution. This allows a better understanding of how faults may occur and the relationship between high-level language constructs and specific fault types. Statistics about fault occurrence per fault type have been collected based on the set of faults observed. These results are important to understand which fault types are representative of residual faults.

2. Proposes an extension to the Orthogonal Defect Classification (ODC) [19], [27] specifically meant for fault injection. In fact, ODC has been successfully used as a tool to improve the software designing process and provides an important basis to understand and classify software faults. However, ODC relates faults to the way faults are corrected, and not to the possible ways they can be emulated. Given the fact that the same faults can be corrected in different ways, a closer look into the exact nature of the faults is necessary for accurate fault emulation, which resulted in a new fault classification that extends ODC for its application in fault injection.
3. Presents the technique G-SWFIT (Generic Software Fault Injection Technique), which is our proposal for a practical approach to inject software faults. This technique emulates the software fault classes most frequently observed in the field through a library of fault emulation operators and injects the faults directly in the target executable code. Each fault emulation operator in the library consists of a machine-code level pattern and the corresponding code changes that represent the software faults most frequently observed in the field for that code pattern. In some sense, our fault emulation operators are similar to mutation operators used in mutation testing [20], [28], [48], [49], but they have three fundamental differences: 1) Our operators work directly in the executable code and not at the high-level source code as was the case of mutations, 2) are based on field observations of real faults and not synthetically generated as it is the case of mutations, and 3) the main goal of our operators is to emulate residual software and not the identification of the best sets of test cases, as was normally the case in software mutation.
4. Evaluates the accuracy of G-SWFIT by comparing the effects of the faults introduced by this technique at the machine-code level with the corresponding high-level faults. In general, we observed a pretty good match between the injected faults and the actual faults that are supposed to be emulated. Section 4 presents these results and use hypothesis testing to evaluate the similarity between faults injected by G-SWFIT and faults inserted at source code level.
5. Discusses the generalization and portability of the proposed method according to multiple aspects, such as the high-level language used in the construction of the target system, the compilers, the compilers optimization options, and the processor architecture. A very important conclusion is that G-SWFIT is practically independent from the high-level language and compiler details, which is a necessary condition to use the technique in systems where the source code is not available.

The structure of the paper is as follows: The next section presents the related research. Section 3 presents the analysis of field data software faults. Section 4 presents the G-SWFIT technique and discusses its accuracy through a fault case study. The important issue of portability is also addressed. Section 5 presents some conclusions.

2 Related Work

The study of software faults has been mainly related to the software development phase, as the software faults originate during the different steps of this phase (requirements, specification, design, coding, testing, etc.). This is an important area of software engineering and many studies have contributed to the improvement of the software development methodologies, with particular emphasis on software testing, software reliability modeling, and software reliability risk analysis [5], [6].

The fault history during the development phase, the operational profile, and other process measures have also been used in software reliability models to estimate the reliability of software and to predict software faults for risk assessment [5], [21], [22], [23], [24].

The study of the software reliability during the operational phase (i.e., after the product deployment) is substantially different from the software under development. The operational environment and the software maturity are different during the operational phase, and the software reliability should be studied in the context of the whole system (and not just in the context of a given application). The difficulties are not only in the instrumentation required to collect data on software faults but also in the fact that the software faults must be analyzed taking into account the system architecture (hardware and software) and not only software modules. Maybe these difficulties account for the fact that the number of works on software faults during the operational phase is lower than the studies available for the development phase. Nevertheless, the study of the effects of actual software faults in the field is of utmost importance for our work, as this is just the kind of fault we want to emulate by fault injection.

Two significant studies of software dependability in Tandem systems are presented in [4], [2]. The impact of software defects on the availability of a large IBM system is presented in [25]. An early study [26] investigated the effect of the workload on the reliability of an IBM operating system based on data collected from field.

An important contribution to promote the collection and study of observed faults is the Orthogonal Defect Classification (ODC) [19], [27]. ODC is a classification schema for

software faults (i.e., defects) in which defects are classified into nonoverlapping attributes and used as a source of information to understand and improve the software product and the software development process. Although ODC is intended to provide feedback to the development process, it also provides a useful defect classification concerning the problem of the emulation of software faults by fault injection (and will be used in our work as a starting point for fault classification).

The idea that code mutations and actual software faults produce similar error patterns and program behavior is supported by the results of an experimental comparison presented in [29]. In fact, according to this study, 85 percent of the errors generated by mutations were also produced by real faults. This suggests that if we improve the accuracy of the fault emulation operators (which use a similar approach to the mutation) by using the outcome of field data on large number of real faults, as we are doing in this paper, we will be able to fill the gap between emulated faults and real faults.

Concerning the area of fault injection, many different approaches have been proposed in the literature. Fault injection has become an attractive approach for experimental evaluation of fault handling mechanisms and assessment of the impact of faults in systems. Fault injection can also help in the estimation of fault-tolerant system measures (e.g., fault coverage and error latency) [26], [30].

The fault injection approaches proposed can either be based on specific hardware, system simulation, or software. Hardware techniques inject physical faults in the target system hardware. Simulation techniques make use of a simulation model of the target system. Finally, a third solution is to emulate hardware faults and errors through software (Software Implemented Fault Injection, or SWIFI for short), which has become very popular as it has low complexity and requires low development effort when compared to the other fault injection approaches. Some examples of SWIFI fault injection tools are Ferrari [31], FTAPE [32], Xception [33], and Goofi [34]. However, all these tools have been proposed for the emulation of hardware faults and their potential in what concerns the emulation of more complex faults such as software faults is very limited.

Other techniques inject faults by corrupting parameters of API calls [35], [36], which has proved to be an effective way to assess the robustness of operating systems. Although the faults injected by these techniques can be considered a possible consequence of software faults (because a software fault can in fact cause wrong API calls), they do not try to directly emulate software faults.

Despite the fact that the emulation of software faults by fault injection remains a largely unknown issue, particularly if the target source code is not available, fault injection has been used in several research works where software faults are the dominant class of faults [37], [38]. In [8], random fault injection (a software testing method to reveal software weaknesses) was used, and other examples of fault injection for software testing purposes can be found in [39], [40].

In [8], fault injection is also proposed for the quantification of the risks created by the software component of a system (experimental software risk assessment). Given the difficulties of knowing which kinds of software faults are most likely to be hidden in the code and their probability of future manifestation, the results of fault injection cannot be used as an absolute measure of risk. Instead, the authors suggest the use of fault injection for the prediction of worst-case scenarios in terms of software risks. This possible use for fault injection is also one of the proposals of our paper, with the difference that instead of injecting random bit-flips, as in [8], we propose the accurate emulation of software faults.

The pragmatic emulation of common programmers' mistakes has been proposed as a "best-effort" approach for emulation of software faults. In [41], faults that try to emulate common programmers' mistakes are used to compare disk and memory resistance to operating systems crashes and, in [9], a similar approach concerning fault representation has been used to improve the design of a reliable write-back file cache.

Previous works from our group have also addressed the injection of software faults. In [42], a large and complex database system was subjected to faults injected by the Xception tool, including faults that may emulate software faults. The faults consist of small changes in the context of the program under execution (e.g., changing a register value, a given memory location content, etc.) triggered by timing or by monitoring processor action. Such faults can indeed emulate some types of software faults, but as the injection is done while the target is actually being executed, the instrumentation level is high and causes some intrusiveness. Furthermore, complex software faults require more elaborate alteration to the program execution and cannot be entirely emulated this way, as shown in [18]. A more elaborate technique to mimic software faults was proposed in [43] and the accuracy of the fault emulation was compared against a small number of real faults (eight faults, actually) showing a reasonably good match. Nevertheless, this technique still has most of the drawbacks of the technique used in [42] as it uses a traditional fault injection tool (Xception).

Faults that emulate typical programmer's mistakes as described in common programming language pitfalls literature were also used in a study of operating systems robustness in the presence of faulty device drivers [16]. The most relevant contribution of this work is that it uses a technique specifically meant for the injection of software faults (and not a traditional SWIFI fault injector), which is the same basic technique used in this paper (but improved with the findings of field data).

The problem of the accurate emulation of software faults by fault injection at the target executable code was first addressed by Christmansson and Chillaregue [15], [7]. These studies propose the same basic set of rules for the generation of errors that emulate software faults. These rules are obtained from the analysis of field data about discovered software faults that have been classified using ODC. In the first paper [15], the rules for error generation for fault injection are proposed for fault forecast, and the second paper [7] addresses the problem of error generation for fault removal.

TABLE 1
Programs Used as Software Fault Sources

Programs	Source location	Description	# faults
CDEX	http://sourceforge.net/projects/cdexos/	CD Digital audio data extractor.	11
Vim	http://www.vim.org	Improved version of the UNIX vi editor.	249
FreeCiv	http://www.freeciv.org	Multiplayer strategy game.	53
pdf2h	http://sourceforge.net/projects/pdftohtml/	pdf to html format translator.	20
GAIM	http://sourceforge.net/projects/gaim/	All-in-one multi-protocol IM client.	23
Joe	http://sourceforge.net/projects/joe-editor/	Text editor similar to Wordstar®	78
ZSNES	http://sourceforge.net/projects/zsnes/	SNES/Super Famicom emulator for x86.	3
Bash	http://cnswww.cns.cwru.edu/~chet/bash/bashtop.html	GNU Project's Bourne Again SHell.	2
LKernel	http://www.kernel.org	Linux kernels 2.0.39 and 2.2.22	93
Firebird	http://sourceforge.net/projects/firebird/	Cross-platform RDBMS engine	2
MingW	http://www.mingw.org/	Minimalist GNU for Windows	60
ScummVM	http://sourceforge.net/projects/scummvm	Iterpreter for adventure engines	74
Total faults collected			668

Christmansson and Chillaregue's papers are seminal papers on this topic, as they give a first contribution to the solution of the problem of accurate emulation of software faults by fault injection. However, the approach proposed by these authors is not general as it can be applied only when there is field data available on previous software faults observed in the desired target system, which is very difficult (or even impossible) to attain in practice. But Christmansson and Chillaregue's work also raises new questions, particularly questions on the accuracy of the emulation and on the adequacy of existing SWIFI tools to perform the injection of errors according to the rules proposed by Christmansson and Chillaregue.

In [18], the approach proposed by Christmansson and Chillaregue was applied using the Xception fault injection tool and it was concluded that only some of the errors defined by the rules proposed in [15] can be injected with a SWIFI tool such as Xception. This motivates the work presented here, which actually merges and expands research results previously presented at DSN and ISSRE conferences [44], [45], [46].

3 Field Study and Software Faults Classification

This section presents the field data on real software faults used in this work. Section 3.1 presents the programs used as sources of real software faults and details the methodology used for the classification of the faults. Section 3.2 presents a first overview of the fault distribution using ODC classification and makes a comparative analysis with the field study done by Christmansson and Chillaregue in [15]. Sections 3.3, 3.4 and 3.5 present the collected faults in detail, according to the classification methodology proposed in Section 3.1.

3.1 Sources of Real Software Faults and Classification Methodology

To address the issue of representativeness, we have collected a large set of real software faults to better understand the exact nature of faults and their occurrence distribution. The source used in our work was a set of *patch* and *diff* files of several open source programs. The source code provided in those files was manually analyzed to understand and classify the faults. A total of 668 faults were analyzed. Table 1 presents a summary of the programs used as sources of real software faults and the links where all the original information (source code, patches, diff files, etc.) about the software faults can be found. It is worth noting that the programs used encompass a broad range of program types: Both user programs (including interactive and command line programs) and operating system (Linux kernels) were used.

The total number of faults collected for each program is dependent on the program age, maturity, and the user community size. Some of the programs (e.g., Bash) are in a mature phase and have few recent faults; other programs (e.g., VIM) are still in the maturation phase and have a large user community that provides many fault reports.

The characterization and classification of software faults is a fundamental step in the definition of a methodology to emulate software faults. The definition of software faults requires the notion of correctness of software. In a broad view, the correctness of software should be measured taking into account the end user/customer needs (expressed by the software requirements). However, for the purpose of this work, it is assumed that the requirements and specification are correct. Thus, a software fault means that the code is not correct somehow (i.e., it does not implement the specification in some particular aspect). More specifically, *the software faults that we would like to emulate by fault injection are the faults originated during the coding phase that have not been detected by the testing procedures* and, consequently, go with the deployed product.

The approach used to classify and to analyze the faults was the following:

1. In a first step, we classified the faults according to the Orthogonal Defect Classification (ODC) [19], [27]. The use of general and well-accepted fault classification is the best way to make our results available for the research community and it allows us to compare our results with previous field studies already available.
2. In a second step, we grouped the faults in each ODC class according to the nature of the defect, defined from a building block programming point of view.

That is, for each ODC class a software fault is characterized by one (or more) *programming language construct* that is either *missing*, *wrong*, or *superfluous*. Programming language constructs may be statements, expressions, function calls, etc. According to this, the *nature* of a software fault can be one of three possible types: **Missing construct**, **Wrong construct**, and **Extraneous construct**. This is particularly relevant when considering fault emulation since emulating an omission (missing construct) is substantially different from emulating a wrong construct (e.g., erroneous expression). As we will see, the fault injection operators proposed in Section 4 are heavily dependent on the nature of the fault.

3. In the last step, faults were further refined and classified in specific types. Each type is defined according to the language construct and program context surrounding its location. This refinement is particularly relevant for fault injection purposes since it helps the identification of suitable locations and the code changes necessary for a given fault type. This final classification can be viewed as an extension to ODC and is used to define fault emulation operators (each operator emulates one specific type of fault).

3.2 ODC Classification and General Analysis

According to the Orthogonal Defect Classification [19], [27], a software fault is characterized by the change in the code that is necessary to correct it (i.e., to make the code consistent with the specification, which is assumed to be correct in our case). The following fault types in ODC are directly related to the code, thus describing software faults as defined in the context of the present work:

1. **Assignment.** Value(s) assigned incorrectly or not assigned at all.
2. **Checking.** Missing or incorrect validation of data or incorrect loop or conditional statements.
3. **Interface.** Errors in the interaction among components, modules, device drivers, call statements, or parameter lists.
4. **Timing/serialization.** Missing or incorrect serialization of shared resources.
5. **Algorithm.** Incorrect or missing implementation that can be fixed by (re)implementing an algorithm or data structure without the need of a design change.
6. **Function.** Affects a sizeable amount of code and refers to capability that is either implemented incorrectly or not implemented at all.

As the available field data does not include any information on timing or serialization properties, we did not consider the Timing/serialization type. The mapping of the faults into one of the remaining ODC types is normally quite straightforward. The Function type, however, required a more detailed analysis of the code in order to figure out whether the correction of the fault has required a redesign or not. In fact, due to the decentralized nature of the software development methodology of the open source community, we don't have direct information on redesign decisions, which has forced us to a more detailed analysis of the faults identified as candidates for the ODC Function type. Table 2 presents the distribution of faults across the five ODC fault types addressed in this work.

TABLE 2
Fault Distribution across ODC Defect-Types

ODC defect-type	# faults	ODC defect-type distribution (%)	
Assignment	143	21.4	(21.98)
Checking	167	25.0	(17.48)
Interface	49	7.3	(8.17)
Algorithm	268	40.1	(43.41)
Function	41	6.1	(8.74)

One interesting topic is the comparison of our results with available field studies that classified real faults using ODC. We compared our fault distribution with the one presented in [15] because that work is the one most closely related to our own. Because that work included Time/Serialization faults, we had to normalize all percentages leaving out that fault type so that a direct comparison could be made. Values presented in [15] are those inside parenthesis in Table 2.

It is worth noting that both our fault distribution and that presented in [15] indicate the same trend in the fault distribution across ODC fault types: Assignment faults have approximately the same weight as Checking faults; Interface and Function faults are clearly the less frequent ones; and Algorithm are the dominant faults. All ODC classes have approximately the same weight in both works. The fact that independent research works obtained a similar fault distribution suggests that this distribution is representative of programs in general. Although more field studies are required to consolidate this conclusion, it seems reasonable that faultloads designed to emulate software faults should take this fault trend into account. Furthermore, the programs analyzed in [15] (large database and operating system code) were quite different from the ones used as source of real faults in our study, which suggests that this fault distribution over ODC types is reasonably independent from the nature of the program (and maybe from the programming language and processor architecture).

The fault distribution observed for each individual program is presented in Table 3. It is important to note that only programs with a significant number of faults (see number of faults in the first row) should be taken into account (nevertheless, we decided to show the results for all the programs). We observed that the programs with a higher number of faults show a similar ODC fault distribution; the only observed deviation was presented by the "Joe" program, which had more checking faults than the global trend. This reinforces the suggestion that software faults do follow a clear pattern of distribution across ODC types.

3.3 Extended Classification and Fault Emulation Discussion

For practical fault injection purposes, the fault types provided by ODC are too broad, meaning that many different faults are encompassed by the same type. Clearly, further refining is needed, not in the sense of an alternative classification but as an additional detail to ODC. As

TABLE 3
Fault Aistribution across ODC Types by Individual Programs

	Programs ▶	CDEX	Vim	FCiv	Pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	FireBird	MingW	ScumVM	Total
	# faults ▶	11	249	53	20	23	78	3	2	93	2	60	74	668
ODC type	Assignment	18.2 %	21.3 %	11.3 %	55 %	4.3 %	25.6 %	66.7 %	100 %	22.6 %	50 %	10 %	24.3 %	21.4 %
	Checking	18.2 %	22.5 %	13.2 %	5 %	52.2 %	44.9 %	0 %	0 %	25.8 %	50 %	38.3 %	8.1 %	25 %
	Interface	54.5 %	6.4 %	7.5 %	0 %	4.3 %	14.1 %	0 %	0 %	5.4 %	0 %	5 %	4.1 %	7.3 %
	Algorithm	9.1 %	44.6 %	52.8 %	40 %	26.1 %	15.4 %	33.3 %	0 %	33.3 %	0 %	46.6 %	56.8 %	40.1 %
	Function	0 %	5.2 %	15.1 %	0 %	13 %	0 %	0 %	0 %	12.9 %	0 %	0 %	6.8 %	6.1 %

TABLE 4
Fault Distribution by Fault Nature

Fault nature	CDEX	Vim	FCiv	Pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	Firebird	MingW	ScumVM	Total	(%)
Missing construct	3	157	35	11	17	34	1	0	63	2	45	61	429	(64.2 %)
Wrong construct	8	85	18	9	6	41	2	2	24	0	14	12	221	(33.1 %)
Extraneous construct	0	7	0	0	0	3	0	0	6	0	1	1	18	(2.7 %)

explained in Section 3.1, we propose to implement this step by analyzing faults from the point of view of the (program) context in which they occur and relate the faults with programming language constructs. From this perspective, a defect is one or more *programming language construct* that is either *missing*, *wrong*, or in *excess* (in this context, a "programming language construct" is any building block of the program: statements, expressions, function calls, etc). According to this idea, we classified each fault according to its nature which can be one of these: **Missing construct**, **Wrong construct**, or **Extraneous construct**. Although this classification is orthogonal to ODC and can be used alone (as is in Table 4), we used it as an extension to ODC fault types to provide a refined view of the faults that is particularly useful for fault emulation.

As we can see in Table 4, faults of the extraneous nature are clearly less frequent than the other two. This is somehow expected, as programmers are more prone to forget to put something in the program or to put it in a wrong way than to insert surplus code. It is interesting to note that missing programming constructs is clearly the dominant type of software bug.

Table 5 presents the total number of missing, wrong, or extraneous faults for each of the five ODC fault types addressed in this work. Some fault examples are also provided to help the reader understand what kind of faults is included in each type (this will be detailed further on). As we can see, for the assignment and interface types, missing program construct faults are less frequent than the wrong construct faults.

Sections 3.3.1, 3.3.1, 3.3.2, 3.3.3, 3.3.4, and 3.3.5 present the occurrences of all faults that were collected according to their nature and source program. Each fault type is assigned a code for easier reference. The C programming language is used in all examples presented.

3.3.1 Assignment Faults

Table 6 shows the results for the detailed analysis of assignment faults. The name given to each fault indicates its exact characteristic. The following explanations are useful:

1. Faults relating to initialization were considered as different faults than general assignment faults.

TABLE 5
Fault Nature Totals across ODC Types

ODC types	Nature	Examples	# faults	% of total
Assignm.	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 brach and loop onstruct (if, while, etc.)	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 param.	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 (e.g. one parameter too many in function call)	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 %
	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 %
	Extraneous	Portions of code were completely superfluous	0	0 %

TABLE 6
Detailed Analysis of Assignment Faults

Fault nature	Specific fault types	CDEX	Vim	FCiv	pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	FireBird	MinGW	ScumVM	Total
Missing construct	Missing variable initialization using a value (MVIV)						10			1			4	15
	Missing variable initialization using an expression (MVIE)						1						1	2
	Missing variable assignment using a value (MVAV)		7	2			2			1			8	20
	Missing variable assignment using an expression (MVAE)		6	1	4	1				4	1		4	21
	Missing variable auto-increment (MVAI)									3				3
	Missing variable auto-decrement (MVAD)			1										1
	Missing *OR sub-expr* in larger expression in assignment (MLOA)													0
	Missing *AND sub-expr* in larger expression in assignment (MLAA)													0
Wrong construct	Wrong parenthesis in logical expr. used in assignment (WPLA)													0
	Wrong logical expression used in assignment (WVAL)													0
	Wrong arithmetic expression used in assignment (WVAE)		7							1		3		11
	Wrong value used in variable initialization (WVIV)	1	2	2										5
	Wrong miss-by-one value used in variable initialization (WVIM)		1											1
	Wrong value assigned to variable (WVAV)	1	9		1				2			3		16
	Miss by one value assigned to variable (WVAM)													0
	Wrong constant in initial data (WIDI)		4					1						5
	Wrong miss-by-one constant in initial data (WIDIM)													0
	Wrong string in initial data (WIDS)		5		6									11
	Wrong string in initial data - missing one char (WIDSL)		5				2							7
	Wrong initial data - array has values in wrong order (WIDM)		1					1						2
	Wrong data types or conversion used (WSUT)		3				4			5				12
Extraneous construct	Extraneous variable assignment using a value (EVAL)						1			1				2
	Extraneous variable assignment using another variable (EVAV)		3							5			1	9
Total faults found		2	53	6	11	1	20	2	2	21	1	6	18	143

2. Missing parts of logical expression (i.e., missing "AND subexpression" or "OR subexpression") were considered different faults from the situations where a logical expression was completely wrong (i.e., different from what it should be).

The fault occurrences in Table 6 are necessarily sparse because some programs have fewer faults than the number of fault types. It is worth noting that a small number of fault types are responsible for most of the fault occurrences.

We decided to use the following empirical criteria to consider a given fault type as being representative of residual faults: 1) The number of occurrences of faults must be at least as high as the average, and 2) the occurrence of faults should not be restricted to only one or two of the programs. The following fault types meet these two requirements: MVIV, MVAV, MVAE, WVAV, WSUT, and EVAV, and these represent the assignment fault specific types that should be addressed in the first place when emulating assignment software faults. We addressed this issue by proposing mutation operators for the emulation of the most representative fault types (Section 4). One interesting aspect is that the most frequent types of faults are in fact reasonably independent from the programming language (e.g., missing initializations and missing assignments seem to be common to any language).

3.3.2 Checking Faults

The occurrences of Checking faults are presented in Table 7. The word "branch" refers to constructs that can affect the execution flow (*if*, *while*, *for*, etc.).

We observed once more that a subset of fault types dominates the fault distribution. For the Checking faults, the types MIA, MLAC, and WLEC are the representative ones. The set of mutation operators presented in Section 4 includes operators for the emulation of faults of these three types.

Fault types MLOC and WPLC were not considered as representative faults because the fault occurrences for these two types are concentrated in only one or two programs: This is more likely a particularity of those two programs and should not be considered as a characteristic for the checking fault type.

TABLE 7
Detailed Analysis of Checking Faults

Fault Nature	Specific fault types	CDEX	Vim	FCiv	pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	FireBird	MinGW	ScumVM	Total
Missing construct	Missing *if* construct around statements (MIA)	1	14	3		5						9	2	34
	Missing "OR EXPR" in expression used as branch condition (MLOC)		15							10		6	1	32
	Missing "AND EXPR" in expression used as branch cond. (MLAC)	1	23	1	1	5	4			7	1	2	2	47
Wrong construct	Wrong parenthesis in logical expr. used as branch condition (WPLC)						31							31
	Wrong logical expression used as branch condition (WLEC)		4	3		2				7		5	1	22
	Wrong arithmetic expression in branch condition (WAEC)													0
Extr. Cons.	Extraneous "OR EXPR" in ixpression used as brach cond (ELOC)											1		1
Total faults found		2	56	7	1	12	35	0	0	24	1	23	6	167

TABLE 8
Detailed Analysis of Interface Faults

Fault nature	Fault specific types	CDEX	Vim	FCiv	pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	FireBird	MinGW	ScumVM	Total
Missing construct	Missing return statement (MRS)						9							9
	Missing parameter in function call (MPFC)		1											1
	Missing *OR sub-expr* in param. of function call (MLOP)													0
	Missing *AND sub-expr* in param. of function call (MLAP)		1											1
Wrong construct	Wrong parenthesis in logical expr. in param. of func. call (WPLP)													0
	Wrong logical expression in param of func. call (WLEP)													0
	Wrong arithmetic expression in param. of func. call (WAEP)		8	3		1							2	14
	Wrong variable used in parameter of function call (WPFV)		2				1			5		3		11
	Wrong value used in parameter of function call (WPFL)		1	1			1						1	4
	Miss by one value in parameter of function call (WPFML)		1											1
	Wrong parameter order in function call (WPFO)		1											1
	Wrong return value (WRV)	6	1											7
Total faults found		6	16	4	0	1	11	0	0	5	0	3	3	49

3.3.3 Interface Faults

The fault distribution for Interface faults is presented in Table 8. We have observed that a very small set of faults dominate the fault occurrence: fault types WAEP and WPFV. Fault type MRS was not considered as representative for interface faults since all the occurrences of its faults are concentrated in only one program.

3.3.4 Algorithm Faults

The occurrences of Algorithm faults are presented in Table 9. Some fault types are somewhat complex and need to be explained.

1. Missing "if (cond) statement(s):" This fault type represents the omission of a block of code made of an if construct and its associated statements which are executed only if the if condition is true.
2. Missing "if (cond)" around statement(s): This fault type consists of having a number of statements always executed instead of only when a condition is met.
3. Missing "if (cond) statement(s) else" before statement(s): This fault type encompasses the situations where one or more statements should exist inside the else part of a missing if-else construct.
4. Missing "if (cond) else statement(s)" around statement(s): This fault type represents the situation of having one or more statements that should be placed inside the if part of a missing if-else construct.
5. Wrong branch/flow construct: The wrong execution flow was used (e.g., an if construct instead a while, etc).
6. Wrong conditional compilation: Faults of this type were made at the preprocessor level and cause wrong portions of the source code (algorithm) to be compiled.
7. Extraneous function call. Only functions calls that were not part of a larger expression were considered as belonging to this type (extraneous function calls existing inside contexts where the return value is needed were considered as falling into the wrong expression fault types).

TABLE 9
Detailed Analysis of Algorithm Faults

Fault nature	Fault specific types	CDEX	Vim	FCiv	pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	FireBird	MinGW	ScumVM	Total
Missing construct	Missing function call (MFC)		28	7	1	1	5			4		2	23	71
	Missing *if* construct plus statements (MIFS)		27	10			1			15		15	12	80
	Missing if-else construct plus statements (MIES)											4	3	7
	Missing *if* construct plus statements plus else before statements (MIEB)	1	10	4		2						1		18
	Missing *if* construct plus *else* plus *statements* around statements (MIEA)									2		1		3
	Missing iteration construct around statement(s) (MCA)		1											1
	Missing case: statement(s) inside a switch construct (MCS)		1											1
	Missing break in case (MBC)		3				1							4
	Missing small and localized part of the algorithm (MLPA)		9		4	2	1			1		5	1	23
	Missing sparsely spaced parts of the algorithm (MLPS)		5		1									6
	Missing large part of the algorithm (MLPL)		3			1		1		3				8
Wrong construct	Wrong function called with same parameters (WFCS)			1			2			6				9
	Wrong function called with different parameters (WFCD)		9	1									3	13
	Wrong branch construct - goto instead break (WBC1)		1	1										2
	Wrong algorithm - small sparse modifications (WALD)		4	1	1									6
	Wrong algorithm - code was misplaced(WALR)		5	3	1									9
	Wrong conditional compilation definitions (WSUC)		1											1
Extraneous construct	Extraneous function call (EFC)		4				2							6
Total faults found		1	111	28	8	6	12	1	0	31	0	28	42	268

TABLE 10
Detailed Analysis of Function Faults

Fault nature	Fault specific types	CDEX	Vim	FCiv	pdf2h	GAIM	Joe	ZSNES	Bash	LKernel	FireBird	MinGW	ScumVM	Total
Missing construct	Missing functionality (MFCT)		3	6						12				21
Wrong construct	Wrong algorithm - large modifications (WALL)		10	2		3							5	20
Total faults found		0	13	8	0	3	0	0	0	12	0	0	5	41

Fault types MFC, MIFS, MIEB, and MLPA are the ones that can be considered representative of Algorithm faults because their total fault occurrences are significant when compared to the average of the other types and its distribution is not restricted to just one or two programs.

3.3.5 Function Faults

Function faults were the least frequent ones. Only the following two types were observed:

1. Missing functionality: This fault relates to functionality that was missing from implementation.
2. Wrong algorithm: This fault type includes the occurrences of algorithms that were not correctly expressed and can only be corrected by a design change. Typically, these faults are not related to particular statements in the code and its correction involves nontrivial code changes.

Table 10 presents the fault occurrence for Function faults.

Both fault types can be considered as representative for function faults since occurrences of its faults are not restricted to only one or two programs and the total number of faults is not significantly below the average.

3.4 Fault Representativeness Analysis

Based on our the observation of the fault occurrence and on the distribution of fault types by the different programs, we have identified a set of fault types that can be considered representative of the most common types of software faults, according to the field data analyzed in this work. It is worth noting that the fault types presented in the previous sections do not represent an exhaustive set of fault types. It is expected that the analysis of more field data on real software faults will bring some other fault types. However, these extra fault types are probably very rare (otherwise, we should have found them among the 668 faults analyzed) and do not change the analysis of the most frequent fault types.

Table 11 presents this "top N" set of fault types. Brief statistical information is also provided. It is worth noting that this set of fault types represent a total of 67.6 percent of all faults collected.

Any effort toward fault emulation must necessarily begin with the fault types that are most representative. In Section 4, we present a set of mutation operators for the emulation of faults of the types identified as representative.

TABLE 11
Fault Coverage of the Representtive Faults Types

Fault nature	Fault specific types	# Faults	ODC types ASG	CHK	INT	ALG	FUN
Missing	*if* construct plus statements (MIFS)	71				✓	
	AND sub-expr in expression used as branch condition (MLAC)	47		✓			
	function call (MFC)	46				✓	
	if construct around statements (MIA)	34		✓			
	OR sub-expr in expression used as branch condition (MLOC)	32		✓			
	small and localized part of the algorithm (MLPA)	23				✓	
	variable assignment using an expression (MVAE)	21	✓				
	functionality (MFCT)	21					✓
	variable assignment using a value (MVAV)	20	✓				
	if construct plus statements plus *else* before statements (MIEB)	18				✓	
	variable initialization (MVIV)	15	✓				
Wrong	logical expression used as branch condition (WLEC)	22		✓			
	algorithm - large modifications (WALL)	20					✓
	value assigned to variable (WVAV)	16	✓				
	arithmetic expression in parameter of function call (WAEP)	14			✓		
	data types or conversion used (WSUT)	12	✓				
	variable used in parameter of function call (WPFV)	11			✓		
Extraneous	variable assignment using another variable (EVAV)	9	✓				
Total faults for these types in each ODC type		452	93	135	25	192	41
Coverage relative to each ODC type (%)		68	65	81	51	72	100

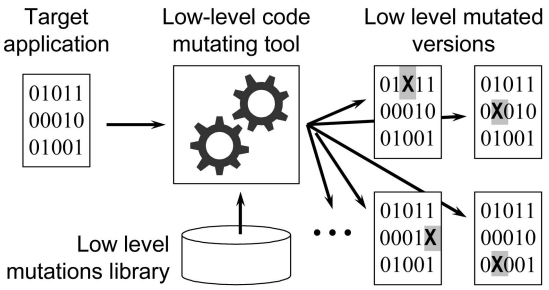

Fig. 2. G-SWFIT key aspects.

4 Emulation of Software Faults

This section describes how faults of the types identified as representative in Section 3.3 can be emulated through the use of the *Generic Software Fault Injection Technique* (G-SWFIT) that was first proposed in [44]. As seen in Section 3, many of the defects that remain in the software after deployment are simple programming errors (their apparent complexity arises from the code that contains them [16], [41]); therefore, techniques such as G-SWFIT, which recreate programmer errors, provide an adequate tool for fault emulation.

Section 4.1 presents a brief description of the key aspects of G-SWFIT. Section 4.2 describes the set of fault emulation operators to be used with this technique that were based on the analysis of the real software faults of Section 3. These fault emulation operators are similar to mutations with the big differences (among others) that they are based on the field study, they represent the most common fault found in the field, and they address the injection at the executable code instead of the source code.

4.1 G-SWFIT Technique

G-SWFIT consists of modifying the ready-to-run binary code of software modules by introducing specific changes that correspond to the code that would be generated by the compiler if the intended software fault were in the high-level source code. A library of fault emulation operators previously defined for the target platform guides the injection of code changes: The target application code is scanned for specific low-level instruction patterns and code changes are performed on the locations identified to emulate the intended high-level faults. This procedure is represented in Fig. 2. The code changing tool comprises the functionality of a disassembler to translate the executable file into assembly code where the machine-code patterns are scanned and the mutations are inserted.

The definition of the fault emulation operator library is based on two principles: the existence of a set of simple high-level programming errors that occur frequently (such as the ones presented in Table 11) and the knowledge of how high-level languages are translated to low-level code, in particular, how high-level programming errors translate into specific low-level instruction sequences.

The G-SWFIT fault emulation operators are defined around the two following concepts:

1. **Search pattern**. This is a sequence of machine code instructions that relate to the high-level constructs where faults can be emulated. Search patterns are defined in such a way that it is possible to identify locations where it is pertinent to emulate a given fault and exclude other locations where such fault would not be meaningful.
2. **Low-level mutation**. This is the code change to apply to a location to emulate of the intended fault.

To assist in definition of the fault emulation operator library, we used a synthetic application containing all the pertinent key high-level constructs and structures, both with and without the considered high-level faults. We did this for the C++ language in a first step but, as presented in Section 4.4, this was generalized to other languages. The observation of the code generated for both cases (with and without faults) allows us to identify the specific machine code patterns where a given class of faults should be emulated by low-level mutations. Using this application with several compilers and different optimization settings, we could understand how such variations affect low level code generation and identify patterns for a given class of faults. These steps are depicted in Fig. 3.

4.2 G-SWFIT Fault Emulation Operators

Our current implementation of G-SWFIT technique supports the fault types listed in Table 12. It is worth noting that these faults alone are responsible for 60 percent of the total number of residual faults, according to our field-data study.

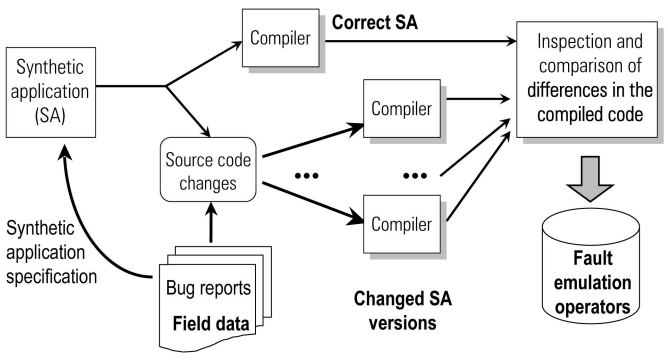

Fig. 3. Opertor definition methodology.

TABLE 12
Fault Emulation Operators

Operator	Fault type addressed
OMFC	MFC - Missing Function call
OMVIV	MVIV - Missing variable initialization using a value
OMVAV	MVAV - Missing variable assignment using a value
OMVAE	MVAE - Missing variable assignment using an expression
OMIA	MIA - Missing if construct around statements
OMIFS	MIFS - Missing If construct plus statements
OMIEB	MIEB - Missing if construct plus statements plus else before statements
OMLAC	MLAC - Missing "AND EXPR" in expression used as branch condition
OMLOC	MLOC - Missing "OR EXPR" in expression used as branch condition
OMLPA	MLPA - Missing small and localized part of the algorithm
OWVAV	WVAV - Wrong value assigned to variable
OWPFV	WPFV - Wrong variable used in parameter of function call
OWAEP	WAEP - Wrong arithmetic expression in parameter of a function call

Due to space restrictions, we cannot present here the implementation details for all the operators. Instead, we include this information in the online Appendix (which can be found at http://computer.org/tse/archives.htm) and present here the implementation of operator OMFC to better convey the notion of how the operators are implemented. Our current operator library is defined for the IA32 platform and the target applications are assumed to be of 32 bit code. The search patterns and its related code changes were derived from the observation of compiler code generation using different optimization settings. Both patterns and code changes are presented in the form of assembly mnemonics according to the Intel notation [47].

Each operator is described according to the rules that define the search pattern and the code change to apply to the locations identified by the search pattern. The search is bound by constraints that help to avoid locations where the code change would not emulate a realistic fault. Note that the same constraint may appear in different operators.

The process of target code analysis is based on a per-module strategy: Each module of the target is analyzed separately from the others. This makes sense as modules are also self-contained units at the source-code level (or at least they should be, according to the programming best practices). A module is typically a subroutine (a function in C, function/procedure in Pascal, etc.).

Modules' starting and ending points relate to very specific instruction patterns and offer useful information

TABLE 13
Module Entry and Exit Points

Module entry point		Module exit point	
Instruction sequence	Explanation	Instruction sequence	Explanation

push ebp
mov ebp, esp
sub esp, *immed*
	stack frame setup	mov esp, ebp
		pop ebp
		ret
			stack frame cleanup

Note: variations may include instructions *enter* / *leave*

TABLE 14
Operator OMFC

Operator	Example	Example with fault	Search pattern	Code change
OMFC	function(…..);	~~function(…..);~~	CALL *target-address*	CALL instruction removed

Contraints	
Return value of the function must not being used	Constraint C01
Call must not be the only statement in the block	Constraint C02

for the code-analysis process (Table 13). More specifically, it is possible to determine the stack frame size (the value "*immed*" in the instruction "sub esp, *immed*"). The size of the stack frame offers hints on how many local variables are used in a particular module (some of the operators require this information).

Operator OMFC (summarized in Table 14) locates function calls in a context where the value returned by the function call (if any) is not used. The removal of a function call from a context where its return value was being used would not represent real software faults. This operator also avoids removing calls in locations where that call is the single statement within its containing block (a comparative example is given in Table 15).

Constraint C01 is implemented by checking that the CALL instruction is not followed by instructions that represent the usage of the returned value. The standard method of returning values from functions is via the EAX register. Thus, it is only necessary to check if the value of that register is used after the call instruction. Any instruction that sets a new value for that register stops this check. If the instruction related to the call is followed by the signature of the exit point of the module, we assume that the return value of that function is being used in a return statement (as in the following: *return function()*;).

Constraint C02 is implemented as follows: In a first step, the full instruction sequence related to the function call is identified (as precisely as possible). This sequence includes the parameter passing before the actual call code and the "add esp, *immed*" instruction to reclaim the stack space used by the those parameters and the use of the return value (if any). It is worth noting that the "*immed*" value in the instruction "add esp, immed" enables the detection of how many parameters were passed to the function being called. In a second step, the boundaries of the code block where the

TABLE 15
Single Statement versus Multiple Statements

Single statement		Not single statement	
Source code	Compiled code	Source code	Compiled code

if (a==123) {
b = function(c);
}
	mov offA[ebp], 123
	cmp offA[ebp], 0
	je *loc-01*
	
	mov eax, off-C[ebp]
	push eax
	call *function-address*
	add esp, 4
	mov off-B[ebp], eax
	loc-01:
		if (a==123) {
		b = function(c);
		c++;
		}
			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:

call is located are identified. The following instructions mark block boundaries: module entry/exit points, unconditional jumps, and conditional jumps to backward locations. Table 15 clarifies this issue with an example of source code and compiled code for a call which is the single statement in its code block and call which is not the single statement within its code block. It is worth noting that the distinction of single statement/nonsingle statement can be adapted to any type of fault location.

The remaining fault emulation operators are implemented in a similar way. We direct the reader to the online Appendix (which can be found at http://computer.org/tse/archives.htm) for the complete list of the operator implementation details.

4.3 G-SWFIT Accuracy Evaluation

In this section, we present an experimental evaluation of the fault emulation accuracy of G-SWFIT. We are particularly interested in two aspects concerning the G-SWFIT fault emulation accuracy:

1. Accuracy of the locations selected by the technique for fault injection. Ideally, the locations selected for fault injection at the executable code are those that directly relate to the locations in the source-code where the intended fault type may realistically exist.
2. Equivalence of the target behavior with the emulated faults (i.e., injected in the executable code) when compared to the behavior caused by real software faults (i.e., inserted at source code level). Ideally, the code changes inserted at the executable code should mimic the instructions that would have been produced by a compiler if the intended faults were indeed present at the source code.

4.3.1 Experimental Setup

The target applications used in the case study were chosen according to the following three rules:

1. The applications should not be too simple, in order to be representative of commonly used programs.
2. The applications must produce a deterministic output from a given input. This enables the automation of the application behavior equivalence testing.
3. Source code should be available. Although this is not necessary for the actual use of G-SWFIT (on the contrary, our main goal is to avoid the need for the source code to emulate software faults), the evaluation of the accuracy presented in the paper needs the source code to compare the effects of low-level mutations with known high-level faults.

The selected applications were the following:

1. **Lzari.** This application is a file compression utility. In order to fully exercise the program algorithm, a variety of input files have been chosen, including both hard to compress data and easy to compress data (e.g., JPEG and text files).
2. **Camelot.** This application produces the minimum number of moves necessary to gather a king and up to 63 knights in the same position of a chessboard. The use of this application provides the additional advantage of having a number of versions each containing a real programming error, which have been used to help the definition of the high-level faults [18].
3. **Gzip.** This application is a widely used compression tool. As Gzip is a real-world commonly used application, its inclusion in this work enables the assurance that G-SWFIT can be used with real-world applications a COTS modules.

TABLE 16
Injected Faults and Executed Runs

Program	Code Level	Faut type								Fault / prog.
		MFC	MIA	MIFS	MLAC	MVAE	MVAV	MVIE	MVIV	
CAMELOT	Source code	15	15	15	2	19	0	0	1	**67**
	Executable code	16	17	17	2	21	0	1	1	**75**
GZIP	Source code	7	11	12	17	10	4	8	2	**71**
	Executable code	7	14	15	17	12	4	9	2	**80**
LZARI	Source code	22	21	21	8	15	0	15	8	**110**
	Executable code	20	24	24	8	18	0	15	8	**117**

4.3.2 Accuracy of Location Selection

The first step of or accuracy evaluation of G-SWFIT is the evaluation of the accuracy of the identification of the locations for fault injection at low-level code when compared to the locations possible for the insertion of the same fault type at the source code level.

We artificially inserted software faults in the source code of the target applications. Faults were inserted in all locations where a given fault type could exist, according to the findings of our field-data study. This means that the number of actual locations used for fault insertion was less than those that the language syntax allowed for. For instance, considering the missing function call fault type, we did not remove a given function call if that call happened to be the only statement in its code block. This restriction was derived from real software faults that we analyzed (note that the same also applies for other "missing" fault types which remove an entire statement, such as a missing variable assignment). Each fault was inserted separately from the other. The result was a set of target mutants, each containing exactly one software fault.

The injection of faults at the target low-level code is an automated procedure carried out by our current G SWFIT implementation. This tool analyzed the target executable code using the instruction patterns defined in the operator library and selected a number of locations for fault injection. The result of this procedure was a number of target mutants, each one containing one software fault. Note that the input given to G-SWFIT is the target in its executable form and not the source code.

Table 16 presents the results of the fault injection process at both source code level and executable code level. Each location in the executable code identified by G-SWFIT was manually inspected and its correspondence with the source-code analyzed. We observed that, with a few exceptions, G-SWFIT was able to accurately identify the locations inside the executable code that correspond to the precise locations in the source code where the intended fault could exist.

We observed that G-SWFIT was able to achieve a perfect match between far fault types MLAC, MVAV, and MVIV for all targets. Additionally, a 100 percent accurate correspondence was also obtained for MFC (GZip) and MVIE (Lzari).

The discrepancies existing between the locations at source code and executable code fall into two categories: more locations in the source code than those in the executable code and more locations in the executable code than those in the source code. We discuss these two cases next.

More locations in the source code than in the executable code. The only occurrence of this situation is related to the missing function call fault type for LZari. The underlying reason for this occurrence is the use of macros in the source code. These particular macros looked like regular functions. To maintain a realistic perspective in our experiment, we considered that if it looks like a function call, it will be eligible for a missing function call fault type. However, the macro expansion for these particular macros in the program does not contain any function call. Therefore, in the executable code, there are two locations less for the MFC fault type.

More locations in the executable code than in the source code. We observed that there are several causes for this type of discrepancy, all of them related to the use of macros. Generally speaking, the use of macros in the source code may cause any type of fault location to be found in the executable code without a direct correspondence in the source code. As an example, we found an occurrence of a MFC fault type location found in the executable code of Camelot that was caused by the existence of a function call inside the code expansion of a macro. Despite the fact that the use of macros may expand to any conceivable portion of code and thus may cause G-SWFIT to find extra locations for any fault type, we observed that there are mainly three cases:

1. Conditional statements inside macro expansion. This relates to the use of macros which have conditional statements inside its expansion code. The emulation operators that may be affected by this are those for MIA and MIFS fault types. In all three programs, G-SWFIT identified more MIA and MIFS fault locations in executable code than in the source-code. By manual inspection we established that the reason for this was indeed the existence of conditional statement in the code of a macro.
2. Assignment inside macro expansion. This is related to the existence of a variable assignment inside the code expansion of a macro. From the programmer viewpoint, the macro may look like a function call (e.g., isspace(c), where c is a variable). G-SWFIT only sees the code generated by the macro and thus any variable assignment occurring inside the expansion code has no counterpart in the source code. The emulation operators that may be affected by this are those for MVAE and MVAV. In our experiments, only MVAE was affected by this (in all three programs).
3. First assignment to variable occurs inside macro-expansion. This is a situation much similar to the previous one. However, it relates to the first assignment to a given variable. Therefore, only MVIE and MVIV fault types may be affected by this issue. In our experiments, only the operator for MVIE fault type was affected (GZIP and Camelot only).

As a conclusion regarding the accuracy of fault location identification, G-SWFIT's main factor for less than 100 percent accuracy is the use of macros. Still, there were several cases where 100 percent accuracy was achieved. The worst case was the fault type MIA in the Gzip program, for which G-SWFIT found 27 percent more fault locations than really exist in the source code. However, this was the worst case of all. Putting all fault types and programs together, we observe that, on average, G-SWFIT found only 9 percent more fault locations and only two locations in the source code were not located in the executable code. This suggests that G SWFIT provides good accuracy concerning fault location identification.

4.3.3 Target Behavior Comparison

The second part of evaluation of G-SWFIT fault emulation accuracy is focused on the target behavior when comparing faults inserted at source code level (that represent real faults in our study) with a fault of the same type injected by G-SWFIT at executable code level.

For each application, a suite of 100 different inputs was provided. Our goal was to provide enough different inputs in order to exercise all parts of the target code. Each version of each target was run against the entire input suite (recall that each target version contains exactly one software fault). The total number of executions was 14,000 for Camelot, 15,700 for Gzip, and 22,700 for Lzari. The host machine was an Intel Pentium III machine running Windows 2000 with 512 Mb of RAM and a 120 Gb HDD (the operating environment provided all the resources for the applications). We used a special starter tool to launch and monitor the exit status of the target applications.

The correctness of the output generated by the target was evaluated using a database of correct results (obtained by running each target with no faults for the entire input suite). In addition to the correctness of the results, other aspects of the application behavior have been used, leading to the following failure modes:

1. Correct behavior ("correct"). The application produced the expected result in the allotted time and did not cause any abnormal event during its execution. Either the injected fault was not activated or it was tolerated by the inherent program redundancy. To minimize the possibility of the injected fault not being activated, a large number of different inputs were used with each version.
2. Incorrect results with error message ("msg"). The application terminated but the results were incorrect and an error message was provided that might help the user.
3. Wrong behavior. The application provided wrong ("wrong") results but no message was provided to alert the user.

TABLE 17
Results for All the Target Versions per Target and per Fault Type

		Camelot						Gzip						Lzari					
		#	#	Results				#	#	Results				#	#	Results			
		fault	runs	Correct	Msg	Wrong	Hang	fault	runs	Correct	Msg	Wrong	Hang	fault	runs	Correct	Msg	Wrong	Hang
MFC	SRC ▶	15	1500	929	0	171	400	7	700	602	50	48	0	22	2200	253	100	1661	186
	Exec ▶	16	1600	929	0	171	500	7	700	602	50	48	0	20	2000	253	100	1461	186
MIA	SRC ▶	15	1500	559	300	462	179	11	1100	603	298	159	40	21	2100	734	379	713	274
	Exec ▶	17	1700	566	298	576	160	14	1400	889	298	173	40	24	2400	1007	675	544	174
MIFS	SRC ▶	15	1500	335	200	486	479	12	1200	909	137	30	124	21	2100	1038	0	573	489
	Exec ▶	17	1700	575	100	346	579	15	1500	1172	74	30	224	24	2400	1277	0	743	380
MLAC	SRC ▶	2	2000	177	0	23	0	17	1700	915	193	492	100	8	800	2	0	0	798
	Exec ▶	2	2000	177	0	23	0	17	1700	477	919	204	100	8	800	2	0	0	798
MVAE	SRC ▶	19	1900	582	0	1318	0	10	1000	384	488	53	75	15	1200	200	48	486	766
	Exec ▶	21	2100	530	0	1070	0	12	1200	548	524	53	75	18	1800	312	48	674	766
MVAV	SRC ▶	0	0	N/A				4	400	111	100	189	0	0	0	N/A			
	Exec ▶	0	0					4	400	111	100	189	0	0	0				
MVIE	SRC ▶	0	0	N/A				8	800	390	326	84	0	15	1500	0	300	108	1092
	Exec ▶	1	0					9	900	537	301	62	0	15	1500	0	300	108	1092
MVIV	SRC ▶	1	100	12	0	88	0	2	200	100	100	0	0	8	800	69	400	201	130
	Exec ▶	1	100	12	0	88	0	2	200	100	100	0	0	8	800	69	400	201	130

SRC = Target executions with source code level faults. Exec = Target executions with G-SWFIT faults

4. The application hanged ("hang"): If the application does not terminate within a timout interval, then it is assumed that it hanged and is terminated by an automated tool. The value of the timeout computed according to the following criteria:

$$timeout = ceil(maxt)^{*}10,$$

where $maxt$ was the maximum time needed by the target to produce the correct result when no fault was present.

According to the notion of program behavior comparison, we are interested in analyzing how similar the distributions of the failure mode occurrences are when a given program/fault type is executed with faults at source-code level and when the same program/fault type is executed with faults injected through G-SWFIT. Table 17 presents the results of the execution of each program/fault type pair. For each of these pairs, we present the failure modes when the program was executed with faults inserted at the source code (lines "SRC") and the failure modes observed when the program was executed with faults injected with G-SWFIT. The number of faults and the number of execution of each program/fault type is also shown ("# faults" and "# runs"). The cases where the behavior of the target was exactly the same for faults in source code and for faults in executable code are highlighted for easier reading.

Considering that MVAV/Camelot, MVAV/Lzari, and MVIV/Camelot could not be evaluated with our experiment (there were no faults to compare), we have a total of 21 different fault type/target combination. Of those, seven (one-third of the total) show a 100 percent accurate behavior; i.e., the target with faults inserted at the source code behaves like the target with faults injected at the executable code. The accuracy of the remaining 14 fault type/target pairs varies, with fault types MIFS and MIA being the fault types that exhibit the greatest discrepancies.

We present the results of the Pearson *goodness-of-fit* and the Crammer association coefficient [50] to better quantify the similarity between the behavior of the program with faults inserted at source code level with the behavior of the same program with faults injected through G-SWFIT. Table 18 presents the results of the null hypothesis test with a significance $a = .05$. The cases where the result is lower than the critical value χ^2 are those where the null hypothesis is not rejected: Camelot/MVAE and LZari/MFC. These cases are highlighted for easier reading. In the remaining cases, we cannot affirm that the distributions of the failures modes are equivalent; however, the test does not show how near/far apart are the failure mode distributions. Table 19 presents the Crammer association coefficients. These coefficients can assume values between 0 and 1; a value close to 0 means that the distributions are strongly associated. If we accept a value lower than 0.1 as representative of similar distributions we now have six more cases of failure mode similarity: MIA/Camelot and Gzip, MIFS/LZari, Camelot/MFC, and LZari/MVAE.

TABLE 18
Pearsons' Goodness-of-Fit Tests

	Camelot	Gzip	Lzari
MFC	7.89	0.00	3.30
MIA	10.42	19.69	151.65
MIFS	126.54	48.04	40.48
MLAC	0.00	730.98	0.00
MVAE	2.49	12.06	27.93
MVAV	N/A	0.00	N/A
MVIE	N/A	21.82	0.00
MVIV	0.00	0.00	0.00

TABLE 19
Crammer Coefficients

	Camelot	Gzip	Lzari
MFC	0.050	0.000	0.028
MIA	0.057	0.088	0.184
MIFS	0.202	0.133	0.094
MLAC	0.000	0.464	0.000
MVAE	0.027	0.074	0.092
MVAV	N/A	0.000	N/A
MVIE	N/A	0.113	0.000
MVIV	0.000	0.000	0.000

The reason behind the less favorable results for MIA and MIFS fault types are related to the facts that 1) more fault locations were identified by G-SWFIT than exist in the source code and 2) the operators for the emulation of those two fault types do not emulate completely the existence of those faults. More specifically, if the expression in a conditional statement being removed has several subexpressions and those subexpressions have side effects, the removal of the side effect of the first subexpression is not emulated—only the decision itself is removed (an example of a condition with side effects: $if\ ((a+=b)>c)$. The side effect is that variable 1 is modified). This limitation is related to our current implementation of G-SWFIT and not a limitation of G-SWFIT itself. The implementation of G-SWFIT is a current project in our research group and we expect to obtain even better accuracy in the future (although we perceive the current results as good concerning fault emulation accuracy).

In the overall picture, our results suggest that G-SWFIT does provide good accuracy when emulating software faults: One third of our cases revealed an exact match in the behavior of the targets; the operators for the MFC, MVAE, and MVIV emulate faults that cause the same behavior in all targets when the same fault types are inserted at the source code; the remaining operators also cause a similar behavior some, but not all, of the targets.

4.4 Generalization and Portability Discussion

Compiler independence and high-level language independence are of key importance for G-SWFIT usefulness in the context of dependability experimentation. In this application field and especially when COTS are involved, it is usual not to have information about the original high-level language and compiler used to build the components under benchmarking. Thus, it is important to investigate how G-SWFIT can be generalized and ported. The following factors are relevant:

1. use of different compilers,
2. use of different compiler optimization settings,
3. use of different high-level languages,
4. different host architectures.

In practice, what is needed to generalize G-SWFIT is to investigate how the factors mentioned above influence the library of low-level instruction patterns and corresponding high-level faults. Obviously, once we have a library of faults, the actual use of G-SWFIT is straightforward, as what is needed is to find the pattern and insert the mutations, which is largely independent from the setup details.

We developed a special synthetic application including all the high-level constructs and code sections where representative faults may occur. The goal of this application was to be used as input to different compilers and enable the comparison of the resulting output code across different compilers (and compiler options). As such, we refer to this application as a synthetic application (because it does not provide useful work per se).

We used the synthetic application on our generalization and portability investigation in the following manner: The application was compiled using compilers, with different optimization settings for each compiler, and compiled for different operating systems and processor architecture. We also ported the application to a different programming language (C++ and Pascal) to observe the differences in the resulting compiled code. The low-level code generated by the different compilers, optimization settings, etc. was then analyzed to confirm if the instruction patterns initially identified were still present or if new patterns have to be added to the library. In the following sections, we discuss our findings.

4.4.1 Different Compilers

To address the issue of different compilers, we analyzed the code generated by different compilers when given the same source code as input. The following compilers were observed for the C language in the Intel/Windows platform: Borland C++, Visual C++, Turbo C++, and GNU C++. It is worth noting that these compilers encompass both recent and old compilers. The inclusion of older compilers addresses the use of legacy COTS and again addresses the possible application scenario when there is no information about which compiler was used to produce a given target.

For all the observed compilers, the code generation is essentially the same (this is due to the existence of compilation standards). Although there are some minor differences in the code produced by those compilers, the consequence of such differences is simply the need to extend the operator library with more instruction patterns and code change definitions to enable the recognition of the code generated by those compilers. We observed that a given pattern produced by one compiler either is also produced by other compilers and is related to the same high-level constructs, or it is not produced at all. Put in other words, we did not observe any case where the same instruction pattern was related to different high-level constructs when using different compilers. This is an important result as it means that operators defined for one compiler remain valid for other compilers. This suggests that the extension of the operator library is enough to encompass different compilers while still maintaining the validity of the previous operators.

4.4.2 Different Optimization Settings

Regarding different optimization settings using the same compiler, all possible settings for the tested compilers (Visual C++, Borland C++, GNU C++, and Turbo C++) were evaluated, including optimizing code for speed and for size. It has been observed that different patterns are generated

for a few high-level constructs, depending on the optimizing settings used. The major differences observed are related to loop unrolling techniques and storing local variables in registers instead of stack memory. The latter could affect the accuracy of operators dealing with variable assignment and initialization (e.g., OAE and OAV). However, such variable optimization is done typically for variables controlling loops. Those variables are already intentionally avoided by those operators' thus, the impact of such optimizations is minimal. The conclusion regarding different optimizing settings is that, although some new low-level instructions patterns have to be considered, none of the new patterns collide with the existing ones, i.e., no ambiguity arises regarding which high-level language construct a given sequence of low-level instruction relates to. As in the case of using different compilers, this issue has as a consequence only the extension of the operator library.

4.4.3 Different Languages

To address the issue of different languages, we analyzed the executable code resulting from C++ and Pascal programs. The programs in the C++ language generate essentially the same low-level code as programs in C when the same kind of high-level construct is used. Although object-oriented constructs were not directly compared, no ambiguity with the previous instruction patterns was detected. To encompass the new high-level faults resulting from the expanded syntax of C++, it is only necessary to extend the operator library to include new instructions patterns and code changes.

In the case of Pascal, the resulting low-level code is also essentially the same as that resulting from C language. This was expected as the two languages are very similar. Some minor differences do exist, for instance, in the way that parameters are passed to functions. However, once again, no ambiguity with previously existing operators was noted.

4.4.4 Different Host Architectures

Regarding the issue of different host architectures, we observed the code generated for the following compiler/platforms: GCC/Linux on an IA32 machine and GCC/OSF Unix for Alpha AXP.

In the case of GCC for Linux over IA32, the code produced is essentially the same as that produced with compilers for Windows over IA32, including the variations caused by different optimization settings. This was somewhat expected since the type of underlying machine is the same. In fact, the differences are related more to the compiler used than to the target OS.

Concerning the GCC for OSF Unix over Alpha AXP, the generated code is completely different. This was expected since there are very few similarities between an Intel 80x86 and an Alpha AXP processor. Therefore, low-level instructions must be also different. However, patterns can still be identified and related with the original high-level constructs. Thus, the underlying idea of the G-SWFIT technique remains valid and applicable. In the case of the Alpha processor, there are specific coding standards, which greatly assist the task of defining low-level instruction patterns. The generalization of different architectures is mainly a question of porting to a different processor, maintaining all the essential aspects of G-SWFIT: An operator library specifying patterns and code changes must be defined prior to the injection of emulated faults, and the injection itself is carried out with ported versions of the same tools used in the case study. Furthermore, it can also be noted that the existing differences between different processors pose no difficulty for the deployment of G-SWFIT since support for different processors families at the same time (i.e., by the same suite of tools) is not expected to be necessary or even particular useful.

The results of our analysis suggest that G-SWFIT is mainly dependent on the target architecture. This means that we need a fault emulation operator library for a given architecture that we want to cover. All the other factors evaluated also have some influence on the operator library; however, this influence consists of having to consider several compilers and optimization settings in order to have a library as complete as possible (meaning that the library contains the relevant instruction patterns and code change definitions).

5 Conclusions

This paper discusses the problem of the emulation of software faults by fault injection. The analysis of field data on more than 650 real software faults shows that simple programmer mistakes are responsible for software failures. Furthermore, there is a clear trend in fault distribution across ODC classes. Other research works using field data also obtained similar fault distributions, in spite of using different sources. This suggests that this fault distribution is a characteristic of software faults in general and should be reflected in the definition of representative faultloads of software faults. Another very important conclusion is that a smaller subset of specific fault types was clearly dominant regarding fault occurrence. These specific fault types are the obvious candidates for the emulation of software faults.

The emulation of software faults can be performed using a technique that changes the low-level code in a way that recreates the code that would have been produced by the compiler if the intended fault had occurred at the high-level source code. G-SWFIT is a fault injection technique specifically conceived for the emulation of software faults according to these ideas. The technique is very simple and can be used even when the original source code is not available. To assist in the definition of mutation operators, faults were classified as missing, wrong, or extraneous constructs. This classification proved to be useful: It is well adapted to common mistakes made by programmers and leads to a considerable simplification in the definition of the mutation operators. Experimental results show that G-SWFIT provide good accuracy when emulating faults.

The generalization and portability of G-SWFIT is dependent mainly on the target architecture, while aspects such as the compiler optimization settings, different compilers, and language used to program the target application only influence the size of the library of emulation operators and the effort needed to generate this library.

Additionally, the concept of injecting software faults in one module with the goal of evaluating the surrounding system can be easily applied to a wide range of system types such as database management systems or Web servers, just to name some examples. Some recent works in dependability benchmarking ([11], [15], [16]) already used G-SWFIT to emulate software faults in order to compare the relative dependability of several important classes of systems.

REFERENCES

[1] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer, "Failure Data Analysis of a LAN of Windows NT Based Computers," *Proc. Symp. Reliable Distributed Database Systems (SRDS-18),* pp. 178-187, 1999.

[2] I. Lee and R.K. Iyer, "Software Dependability in the Tandem GUARDIAN System," *IEEE Trans. Software Eng.,* vol. 21, no. 5, pp. 455-467, May 1995.

[3] M. Sullivan and R. Chillarege, "Comparison of Software Defects in Database Management Systems and Operating Systems," *Proc. 22nd IEEE Fault Tolerant Computing Symp. (FTCS 22),* pp. 475-484, July 1992.

[4] J. Gray, "A Census of Tandem Systems Availability between 1985 and 1990," *IEEE Trans. Reliability,* vol. 39, no. 4, pp. 409-418, Oct. 1990.

[5] J. Musa, *Software Reliability Engineering.* McGraw-Hill, 1996.

[6] M.R. Lyu, *Handbook of Software Reliability Engineering.* IEEE Computer Society Press & McGraw-Hill, 1996.

[7] 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," *Proc. Seventh IEEE Int'l Symp. Software Reliability Eng. (ISSRE '96),* Oct./Nov. 1996.

[8] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting How Badly 'Good' Software can Behave," *IEEE Software,* vol. 14, no. 4, 1997.

[9] W.T. Ng and P.M. Chen, "Systematic Improvement of Fault Tolerance in the RIO File Cache," *Proc. IEEE Fault Tolerant Computing Symp. (FTCS),* June 1999.

[10] A. Brown and D. Patterson, "Toward Availability Benchmark: A Case Study of Software RAID Systems," *Proc. USENIX Ann. Technical Conf.,* pp 263-276, June 2000.

[11] M. Vieira and H Madeira, "A Dependability Benchmark for OLTP Application Environments," *Proc. 29th Int'l Conf. Very Large Databases (VLDB '03),* Sept. 2003.

[12] J. Zhu, J. Mauro, and I. Pramanick, "R3—A Framework for Availability Benchmarking," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '03),* pp. B86-B87, 2003.

[13] S. Lightstone, J. Hellerstein, W. Tetzlaff, P. Janson, E. Lassettre, C. Norton, B. Rajaraman, and L. Spainhower, "Toward Benchmarking Autonomic Computing Maturity," *Proc. First IEEE Conf. Industrial Automatics (INDIN '03),* Aug. 2003.

[14] K. Kanoun, J. Arlat, D. Costa, M.D. Cin, P. Gil, J.C. Laprie, H. Madeira, and N. Suri, "DBench: Dependability Benchmarking," *Proc. Supplement of the IEEE/IFIP Int'l Conf. Dependable Systems and Networks (DSN '01),* 2001.

[15] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults," *Proc. 26th IEEE Fault Tolerant Computing Symp. (FTCS 26),* pp. 304-313, June 1996.

[16] J. Durães and H. Madeira, "Characterization of Operating Systems Behavior in the Presence of Faulty Device Drivers through Software Fault Emulation," *Proc. Pacific Rim Int'l Symp. Dependable Computing (PRDC '02),* pp. 201-209, 2002.

[17] J. Durães, M. Vieira, and H. Madeira, "Web Server Dependability Benchmaking," *Proc. Conf. Computer Safety (SAFECOMP),* 2004.

[18] H. Madeira, M. Vieira, and D. Costa, "On the Emulation of Software Faults by Software Fault Injection," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '00),* pp. 417-426, 2000.

[19] R. Chillarege, "Orthogonal Defect Classification," *Handbook of Software Reliability Eng.,* chapter 9, IEEE CS Press, McGraw-Hill, 1995.

[20] R. DeMillo, D. Guindi, W. McCracken, A. Offut, and K. King, "An Extended Overview of the Mothra Software Testing Environment," *Proc. ACM SIGSOFT/IEEE Second Workshop Software Testing, Verification, and Analysis,* pp. 142-151, July 1988.

[21] S. Yamada, M. Ohba, and S. Osaki, "S-Shaped Software Reliability Growth Models and Their Application," *IEEE Trans. Reliability,* vol. 33, no. 4, pp. 289-292, Oct. 1984.

[22] J.K. Chaar, M.J. Halliday, I.S. Bhandari, and R. Chillarege, "In-Process Evaluation for Software Inspection and Test," *IEEE Trans. Software Eng.,* vol. 19, no. 11, pp. 1055-1070, Nov. 1993.

[23] T. Khoshgoftaar et al. "Process Measures for Predicting Software Quality," *Proc. High Assurance Systems Eng. Workshop (HASE '97),* 1997.

[24] J.P. Hudepohl et al., "EMERALD: A Case Study in Enhancing Software Reliability," *Proc. IEEE Eighth Int'l Symp. Software Reliability Eng. (ISSRE '98),* pp. 85-91, Nov. 1998.

[25] M. Sullivan and R. Chillarege, "Software Defects and Their Impact on Systems Availability—A Study of Field Failures on Operating Systems," *Proc. 21st IEEE Fault Tolerant Computing Symp. (FTCS 21),* pp. 2-9, June 1991.

[26] R.K. Iyer, "Experimental Evaluation," *Proc. 25th IEEE Symp. Fault Tolerant Computing (FTCS 25),* pp. 115-132, June 1995.

[27] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D. Moebus, B. Ray, and M. Wong, "Orthogonal Defect Classification—A Concept for In-Process Measurement," *IEEE Trans. Software Eng.,* vol. 18, no. 11, pp. 943-956, Nov. 1992.

[28] T. Budd, "Mutation Analysis: Ideas, Examples, Problems, and Prospects," *Computer Program Testing,* pp. 129-134, 1981.

[29] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *Proc. Third Symp. Software Testing and Analysis (ISSTA '03),* pp. 158-171, Jan. 1996.

[30] J. Arlat et al., "Fault Injection and Dependability Evaluation of Fault Tolerant Systems," *IEEE Trans. Computers,* vol. 42, no. 8, pp. 919-923, Aug. 1993.

[31] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," *Proc. 22th IEEE Fault Tolerant Computing Symp. (FTCS 22),* pp. 336-344, June 1992.

[32] T. Tsai and R.K. Iyer, "An Approach to Benchmarking of Fault-Tolerant Commercial Systems," *Proc. 26th IEEE Fault Tolerant Computing Symp. (FTCS 26),* pp. 314-323, June 1996.

[33] J. Carreira, H. Madeira, and J.G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," *IEEE Trans. Software Eng.,* vol. 24, no. 2, Feb. 1998.

[34] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Goofi: Generic Object-Oriented Fault Injection Tool," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '01),* July 2001.

[35] J.-C. Fabre, F. Salles, M. Moreno, and J. Arlat, "Assessment of COTS Microkernels by Fault Injection," *Proc. Seventh IFIP Working Conf. Dependable Computing for Critical Applications (DCCA '99),* pp. 25-44, 1999.

[36] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing Operating Systems Using Robustness Benchmarks," *Proc. 16th Int'l Symp. Reliable Distributed Systems (SRDS '97),* pp. 72-79, 1997.

[37] J. Hudak, B. Suth, D. Siewiorek, and Z. Segall, "Evaluation and Comparison of Fault-Tolerant Software Techniques," *IEEE Trans. Reliability,* vol. 42, no. 2, pp. 190-204, June 1993.

[38] W. Kao, "Experimental Study of Software Dependability," PhD thesis, Technical Report CRHC-94-16, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Illinois, 1994.

[39] J. Bieman, D. Dreilinger, and L. Lin, "Using Fault Injection to Increase Test Coverage," *Proc. Seventh IEEE Int'l Symp. Software Reliability Eng. (ISSRE '96),* Oct. /Nov. 1996.

[40] D. Blough and T. Torii, "Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message-Passing Parallel Computers," *Proc. 27th IEEE Fault Tolerant Computing Symp. (FTCS 27),* pp. 258-267, June 1997.

[41] W.T. Ng, C.M. Aycock, and P.M. Chen, "Comparing Disk and Memory's Resistance to Operating System Crashes," *Proc. Seventh IEEE Int'l Symp. Software Reliability Eng. (ISSRE '96),* 1996.

[42] D. Costa, T. Rilho, and H. Madeira, "Joint Evaluation of Performance and Robustness of a COTS DBMS through Fault-Injection," *Proc. Dependable Systems and Networks Conf. (DSN '00),* June 2000.

[43] D. Costa, T. Rilho, M. Vieira, and H. Madeira, "ESFFI—A Novel Technique for the Emulation of Software Faults in COTS Components," *Proc. Eighth Ann. IEEE Int'l Conf. Eng. of Computer-Based Systems (ECBS '01),* Apr. 2001.

[44] J. Durães and H. Madeira, "Emulation of Software Faults by Selective Mutations at Machine-Code Level," *Proc. 13th IEEE Int'l Symp. Software Reliability Eng. (ISSRE '02),* Nov. 2002.
[45] J. Durães and H. Madeira, "Definition of Software Fault Emulation Operators: A Field Data Study," *Proc. IEEE/IFIP Int'l Conf. Dependable Systems and Networks, Dependable Computing, and Comm. (DSN-DCC '03),* June 2003.
[46] J. Durães and H. Madeira, "Generic Faultloads Based on Software Faults for Dependability Benchmarking," *Proc. IEEE Int'l Symp. Dependable Systems and Networks (DSN '04),* 2004.
[47] *Intel Architecture Software Developer's Manual, Vol. 2: Instruction Set Reference,* Intel, 1997.
[48] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs against Errors.* John Wiley and Sons, 1997.
[49] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* vol. 11, no. 4, pp. 34-41, Apr. 1978.
[50] R.E. Kirk, *Statistics: An Introduction,* fourth ed. Assessment Systems Corp., 1999.

João A. Durães received the BS, MSc, and PhD degrees in informatic engineering from the University of Coimbra in 1994, 1999, and 2006, respectively. During 1994-2006, he has been with the Centre for Informatics and Systems of the University of Coimbra as a researcher. He has been teaching computer-related courses at the Institute of Engineering of Coimbra since 1995. He was the recipient of the IEEE/IFIP William C. Carter award for the best paper at the Dependable Systems and Networks Symposium (DSN 2003). He is a member of the IEEE.

Henrique S. Madeira is an associate professor at the University of Coimbra, where he has been involved in research on dependable computing since 1987. He has authored or coauthored more than 100 papers in refereed conferences and journals and has coordinated or participated in tens of projects funded by the Portuguese government and by the European Union. He was the vice-chair of the IFIP Working Group 10.4 Special Interest Group (SIG) on Dependability Benchmarking from the establishment of the SIG from the summer of 1999 until 2002. He was the program cochair of the International Performance and Dependability Symposium track of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-PDS '04) and was appointed Conference Coordinator of IEEE/IFIP DSN '08. He is currently the president of the Centre for Informatics and Systems of the University of Coimbra Research Centre and the head of the Department of Informatics Engineering at the University of Coimbra. He is a member of the IEEE.