Injection of faults at component interfaces and inside the component code: are they equivalent?

R. Moraes[1], R. Barbosa[2], J. Durães[3], N. Mendes[3], E. Martins[1], H. Madeira[3]

[1]State University of Campinas, UNICAMP, São Paulo, Brazil

[2]Critical Software SA, Coimbra, Portugal

[3]CISUC, University of Coimbra, Portugal

{regina@ceset, eliane@ic}.unicamp.br, rbarbosa@criticalsoftware.com, jduraes@isec.pt, {naaliel, henrique} @dei.uc.pt

Abstract

The injection of interface faults through API parameter corruption is a technique commonly used in experimental dependability evaluation. Although the interface faults injected by this approach can be considered as a possible consequence of actual software faults in real applications, the question of whether the typical exceptional inputs and invalid parameters used in these techniques do represent the consequences of software bugs is largely an open issue. This question may not be an issue in the context of robustness testing aimed at the identification of weaknesses in software components. However, the use of interface faults by API parameter corruption as a general approach for dependability evaluation in component-based systems requires an in depth study of interface faults and a close observation of the way internal component faults propagate to the component interfaces. In this work we present the results of experimental evaluation of realistic component-based applications developed in Java and C using the injection of interface faults by API parameter corruption and the injection of software faults inside the components by modification of the target code. The faults injected inside software components emulate typical programming errors and are based on an extensive field data study previously published. The results show the consequences of internal component faults in several operational scenarios and provide empirical evidences that interface faults and software component faults cause different impact in the system.

1. Introduction

Fault injection techniques have been extensively used to evaluate specific fault tolerance mechanisms and to assess the impact of faults in systems and have progressively been adopted by the computer industry [1]. In general, the types of the faults injected emulate hardware transient faults. However software faults are currently more relevant than hardware faults as software is becoming extremely complex and the most common types of faults discovered in deployed systems are in fact software faults.

The problem of emulation of software faults by fault injection has been addressed in the last decade [8, 9, 22, 27], leading to practical techniques that emulate software faults with acceptable accuracy [11, 12].

An alternative to the actual injection of software faults inside components is the emulation of its effects by injecting exceptional or invalid values at the components interface. In this context it is assumed that these exceptional or invalid values represent in fact the possible consequences of real software faults in the preceding components/programs (i.e. faults in the component that requires some information through an API call and sends parameters to the target component). Although this approach has provided useful results concerning the exposure of hidden robustness weaknesses and how to remedy them (e.g., through wrappers), the issue of the representativeness of the values injected relative to effects of real residual faults is still an important issue, particularly in contexts of dependability benchmarking.

It is worth noting that the use of exceptional or invalid values at the system/component API interface has been largely used in the context of robustness testing. In fact, robustness testing has been used with remarkable success to expose robustness weaknesses of operating systems [13, 18, 19, 26], and it relies on a simple and logical argument: if a faulty program invokes another (e.g., the operating system through its API) with abnormal parameters, the invoked program should behave in a robust way and not crash in any circumstances. In this context, fault representativeness is not an issue in classical robustness testing, as this kind of testing does not specifically require the emulation of realistic software faults. In this paper we do not evaluate the equivalence of the robustness testing to the effects of actual residual faults. Instead, we compare the impact of interface faults with the effects caused by faults injected inside the component.

Although started as a robustness testing approach, the idea of injecting faults at interface level has been generalized to be applied to fine grain software components (and not just to operating systems). This generalization of robustness testing is often called as component interface fault injection and has been used to evaluate the impact of faulty components (simulated by erroneous values passed to subsequent components) in the rest of the system [24, 25, 30]. Additionally, component interfaces faults also emulate component incompatibilities, such as appropriated components that can not interoperate [32].

Unlike what happens to robustness testing, the use of interface fault injection as a general approach for component-based systems evaluation raises the question of knowing *whether the faults injected at interface level do represent possible consequences of residual software bugs in preceding components*. Unfortunately, the mapping between component software faults and interface faults is not clear, which means that the representativeness of interface fault injection experiments may be questionable. To the best of our knowledge there are no works that address the possible equivalence between software faults in a given component and the interface faults injected in the outgoing calls from that component to the remaining system.

This work addresses this problem by observing the visible effects of realistic software faults in a given component, as they may appear at interface level of other components. This experiment gives a clear contribution to the knowledge about interface faults by identifying how software fault injection can guide interface fault injection (and even robustness testing) experiments. Our study also contributes to a better understanding of error propagation in component based system, which has been a central issue in many recent researches (e.g., [14, 17]).

The paper is organized as follows: Section 2 provides a short overview of fault injection at component interface level and injection of software faults inside the components. The experimental setup and the tools that were used in the experiments are presented in Section 3. Section 4 presents and discusses the results obtained and Section 5 concludes the paper and outlines our future work.

2. Two ways of Emulating Software Faults

Software components may contain faults and are exposed to faults from the surrounding environment (other components and systems). Therefore, a major challenge for designers and programmers is to assess how sensitive the system is to faults in any of its software components. Internal and interface faults can be correlated, at least in a partial way, as interface faults can represent the error propagation caused by an internal fault from the preceding components.

The emulation of internal faults may conceivably be achieved through the injection of interface faults, which is relatively inexpensive to develop and execute, as the fault injection mechanism would use the normal component interfaces. However, this assumption requires that the faults injected at the interface level must present the same impact of the errors propagated by the faults that may realistically exist inside the preceding components.

Another approach for the emulation of software faults is the actual modification of the target code in order to inject software faults defined according the most frequent types of real software faults found in field studies (e.g., [12]). This approach offers a better representativeness as the modification injected in the code reproduces the instruction sequences that would be present in the target code if the intended fault were indeed there in the first place (i.e., if a given programming error were indeed present at the target source code). The reproduction of the programming errors directly at the executable code level presents technical challenges and the modification of the target code may be not acceptable in several evaluation contexts [20].

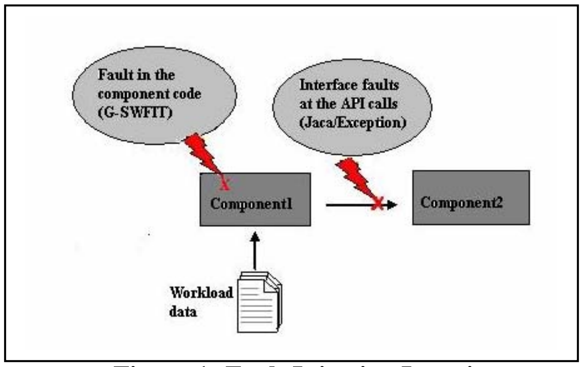

Fault in the component code (G-SWFIT)
Interface faults at the API calls (Jaca/Exception)
Component1
Component2
Workload data

Figure 1: Fault Injection Location

It is worth noting that internal component faults are used in the present work to provide a better understanding on how interface faults can be compared to the injection of software faults, and how both techniques can be used together. Thus, the modification of the target code is not an issue concerning the goals of the present paper. Figure 1 presents a schema to explain where internal and interface faults are injected.

2.1. Injection of Software Faults

Few studies have addressed the problem of injection of software faults, especially when compared to the vast literature on injection of hardware faults. Fault injection using simple fault models has been used with success in several research works where software faults are the most relevant class of faults. Several examples of software weaknesses revealed by faults injected at random can be found in [30]. Other examples of fault injection used for software testing can be found in [2, 3]. Software fault injection emulated through simple program code corruption was also used to validate fault tolerance mechanisms [28].

The problem of injecting representative software faults was first addressed in [8]. That work was done in the context of IBM's Orthogonal Defect Classification (ODC) project [7] and the proposed method requires field data about real software faults in the target system or class of target systems. This requirement (the knowledge of previous faults) greatly reduces the usability of the method, as this information is seldom available and is simply not possible to obtain for new software. Furthermore, as shown in [22], typical fault injection tools are not able to inject a substantial part of the types of faults proposed in [8].

To the best of our knowledge, the first practical approach to inject software faults was proposed in [11]. The approach is based on a technique named Generic Software Fault Injection Technique (G-SWFIT) and is supported by the findings from a field study on real software faults in a variety of programs [12]. In order to better emulate software faults by fault injection G-SWFIT uses an extension of Orthogonal Defect Classification (ODC) [7] and classifies faults according to the point of view of the program context in which they occur, and closely relates faults with programming language constructs. According to this idea, a software defect is one or more programming language constructs that are either **missing**, **wrong** or **in excess**. The resulting fault types describe the morphology of software faults according to the language constructs where they can realistically appear (which helps the definition of code changes that must be injected to emulate each fault type), and relates the faults with the surrounding program context (which greatly improves the selection of appropriates locations for fault injection).

Based on the field study presented in [12], and on field data from IBM's ODC project [8], the G-SWFIT software fault injection technique uses a library of fault emulation operators that represent the most common type of faults observed in the field. The code changes actually injected with G-SWFIT represent the code that would have been generated by the compiler if the intended software faults were in fact in the high level source code. Both the code pattern and the code changes were defined based on an extensive field data study in real software faults discovered in deployed software [12].

The G-SWFIT library of fault emulation operators was established for executable code. This means that G-SWFIT can be used in components even when we do not have the component source code (obviously, G-SWFIT can also be used at source code level, whenever the source code is available). The G-SWFIT operator library also considers the different aspects that have impact on the code structure and on the type of bugs it supports, such as different programming languages, compilers and compilers optimization options, which means that the technique is portable.

G-SWFIT is based on a two steps methodology. The fault locations are identified in the first step, resulting in the set of faults to be injected. This step occurs before the actual experimentation. The faults are actually injected in step two during the target execution in a very simple and low intrusive task, as each fault location have been previously identified in step one. The result of the scan process is a map of the target identifying the locations suitable for the emulation of specific fault types.

2.2. Injection of Interface Faults

The injection of interface errors attempts to emulate the consequences of errors that are propagated from faulty components (as mentioned, in robustness testing contexts this emulation requirement is often relaxed). The values injected are normally based on the semantics of the system/component interface and in the data type used in the interface. Typical values used are extreme values of the data type used in the interface or valid and invalid values according to the interface semantics (e.g. Ballista project [18]).

Interface faults can be injected directly at the interface between components to simulate the situation where a component fails and outputs corrupted information to the other components. The fundamental assumption behind this methodology in component-based systems is that the parameter corruptions represent a plausible consequence of real residual software faults existing in the software component that calls the API. However there is no guarantee that the values being injected really represent real residual faults produced by the API call. Even if some of the values injected can be produced by an internal fault, there is no assurance that such internal fault is a representative one.

Works about interface fault injection used in the robustness testing context are relatively abundant. BALLISTA is well-known tool aimed at the automated robustness testing [18, 21], which submits the target interface with valid and invalid inputs and can be interpreted as emulating the behavior of an erroneous module calling the target API.

The DTS tool [29] is a robustness testing tool that injects faults in the parameters of the calls to Windows NT system libraries. Faults are injected during actual calls performed by real programs during run-time. The RIDDLE tool [15, 16] is a robustness benchmark which uses a combination of random input, malicious input and boundary values to test the robustness of black-box components. MAFALDA [14] is a tool aimed at the evaluation of microkernel behavior in the presence of faults that uses system-call parameter corruption specifically aimed at the emulation of the effects of residual faults existing at the application level. A robustness benchmark tool based on the corruption of the parameters to the operating systems calls is presented in [10]. This tool uses a tailored workload to specifically call the OS services using invalid parameters. The injected values were defined based on the semantic of the target OS call.

Xception is a set of tools aimed at the experimental dependability evaluation [5]. Xception can inject interface faults for several processor architectures and it includes a set of modules to assist the execution of fault injection campaigns and information collection. The fault model supports the traditional robustness testing concept and includes a broad range of data types.

The Jaca tool [23] is used to inject interface faults in object-oriented systems written in Java programming language. Jaca uses reflective mechanisms of the Java language to inject the faults without changing the target system structure and is able to monitor the target system to examine the data flow across components without actual injecting faults. Jaca uses the Javassist reflection toolkit [6] to perform instrumentation at the bytecode level and does not require the source code of the target system. Current version of Jaca can operate with the public interface of classes by altering values of attributes, method parameters and return values.

3. Experimental setup and methodology

To understand the equivalence (or differences) between faults injected in the component code and faults injected at the component interface, we injected faults according to both methodologies and compared the impact of the injected faults in the target system (characterized through the well-known notion of failure mode). From a statistical point of view, faults injected at interface-level should have similar consequences as the faults injected in the component code.

In addition to the fault impact on the system, we also analyze the values that are passed in the interface calls after the injection of an internal fault with the values used when injecting faults at the interface. This analysis is aimed at the understanding of the relation between the faults injected through the interface and realistic faults existing in the preceding components.

We used two different experimental setups: the first is an object oriented database application written in Java (Ozone) and the second is a real-time application written in C. Each setup involves the use of component-based software system so that there is a clear boundary between modules (this a necessary requirement for the injection of the interface faults, as well as for the delimitation of the code where faults are injected). Both experimental setups were subjected to both types of fault injection.

We used G-SWFIT to inject faults in the component code. Jaca and Xception were used to inject faults at the interface. This selection was guided by the fact that these tools/techniques were developed in the context of previous works from our group and are well understood technologies.

We describe the experimental setups in the following sub-sections.

3.1. Ozone/OO7 experimental setup

This setup is composed by the object-oriented database system Ozone 1.1 and the Wisconsin OO7 benchmark acting as the database workload. The OO7 benchmark is an object-oriented database application inspired in the notion of a design library composed by a set of Computer Aid Software Engineering (CASE) documents organized hierarchically. This setup runs on top of the 1.5 Java runtime environment. As a benchmark, Wisconsin OO7 can be considered as a representative application for object-oriented systems and implements the main functionalities expected from such kind of applications [4], and can be applied on any object-oriented database management system.

This system is perceived as a software system composed of two large components which are the Ozone and the OO7. In this scenario the target component for the injection of *faults in the code* (i.e., internal faults) is the OO7. Faults were injected directly into its OO7 classes using G-SWFIT. *Interface* *faults* were injected at the interface of the Ozone classes whose public methods are invoked by the OO7 code. The injection of interface faults was carried out using the Jaca tool (see Figure 2). The observation of the behavior of the OO7 and the Ozone, as well as the database state, allows us to compare the effects of each type of faults and understand the similarity of interface faults regarding the faults existing in the code of the components.

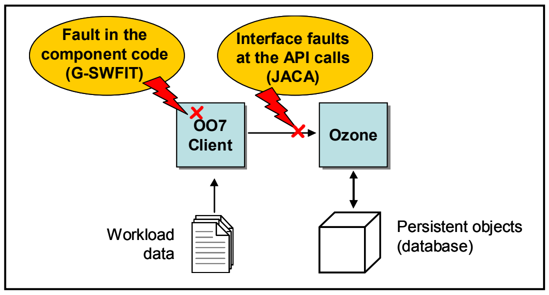

Fault in the component code (G-SWFIT)
Interface faults at the API calls (JACA)
OO7 Client
Ozone
Workload data
Persistent objects (database)

Figure 2: Ozone/OO7 Setup Overview.

3.2. Data-Handling/RTEMS experimental setup

This setup represents a real-time space application based on real-mission project requirements from the European Space Agency (ESA). This setup consists of a Command and Data Management System (CDMS) responsible for the management of the on-board systems and communication between the spacecraft and the ground control. It was implemented in C and runs on top of the Real Time Executive for Multiprocessor Systems (RTEMS) operating system. It is composed by the following components/subsystems:

- **Packet Router (PR):** This subsystem is composed by two distinct components:
  - **Telecommand Manager (TC):** responsible for the reception of ground commands and redirecting to the correct sub-system.
  - **Telemetry Manager (TM):** responsible for selectively sending telemetry to ground.
- **Power Conditioning System (PCS):** responsible for management of the power sources and the power circulation in the spacecraft.
- **On Board Storage (OBS):** responsible for storing telemetry to be sent to ground.
- **Data Handling System (DHS):** responsible for managing all data transactions between ground systems and spacecraft.
- **Reconfiguration Manager (RM):** responsible for recovering the spacecraft from failures.
- **Payload (PL):** responsible for performing science related activities, in this case, controlling the telescope and acquiring images.
- **Hardware Abstraction Layer:** responsible for handling low level communications, namely, controlling the communications devices;
- **Hardware Communication Module:** responsible for passing the command and telemetry to and from the hardware abstraction layer;
- **System Startup Module:** responsible for system initialisation and operations start up.

The workload executed in this setup is a mission scenario where a simulated space telescope is being controlled. The system controls the telescope parameters (aperture, exposition time, etc.), collects data and sends it to ground system. All data involved in this scenario is predefined which allows deterministic experiments. Also, independent mission scenario runs will produce exactly the same result, which allows further determinism. The starting point of the workload is an acknowledgement command sent from the CDMS to the ground control. After that, the ground control sends a series of commands for the CDMS to adjust telescope settings and capture image. For each command sent the CDMS sends back telemetry information. The timing of the commands and the contents of the telemetry information are used to detect the system correctness/failure during the experiments.

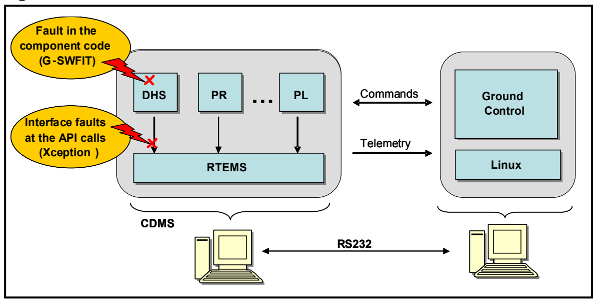

Fault in the component code (G-SWFIT)
Interface faults at the API calls (Xception)
DHS
PR
...
PL
RTEMS
Commands
Telemetry
Ground Control
Linux
CDMS
RS232

Figure 3: Data Handling/RTEMS Setup Overview.

As shown before, CDMS is divided in several components. The components used in the mission scenario were selected for the injection of *fault in the code* (G-SWFIT). In other words, in this setup we are comparing the effects of faults in the selected components with the effects of faults injected at the interface of the components used by these selected components. All the explicit interaction of this component with remaining system is made via the operating system services. Thus, the target for *interface faults* can only be the OS.

The ground control software is hosted in a computer running Linux. CDMS runs on a X86 architecture based PC on top of the RTEMS kernel and is connected to the ground system via a RS232 link. The interface faults are injected with the Xception tool [5]. The faults in the CDMS code are injected with the aid of G-SWFIT. Figure 3 illustrates the key aspects of this setup.

Table 1 – Representativity of the fault types included in the faultload

Fault types	Description	Perc. Observed in field study	ODC classes
MIFS	Missing "If (*cond*) { statement(s) }"	9.96 %	Algorithm
MFC	Missing function call	8.64 %	Algorithm
MLAC	Missing "AND EXPR" in expression used as branch condition	7.89 %	Checking
MIA	Missing "if (*cond*)" surrounding statement(s)	4.32 %	Checking
MLPC	Missing small and localized part of the algorithm	3.19 %	Algorithm
MVAE	Missing variable assignment using an expression	3.00 %	Assignment
WLEC	Wrong logical expression used as branch condition	3.00 %	Checking
WVAV	Wrong value assigned to a value	2.44 %	Assignment
MVI	Missing variable initialization	2.25 %	Assignment
MVAV	Missing variable assignment using a value	2.25 %	Assignment
WAEP	Wrong arithmetic expression used in parameter of function call	2.25 %	Interface
WPFV	Wrong variable used in parameter of function call	1.50 %	Interface
	Total faults coverage	**50.69 %**	

4. Experimental results and discussion

An important aspect of the experiments is the definition of the set of faults to inject. We begin this section by presenting the details of the faults to inject in the target code (sub-section 4.1), and the details of the interface faults (sub-section 4.2).

4.1. Set of faults in the target component code.

The emulation of software faults in the target code requires the precise knowledge of which faults are to be injected: different faults types will require different target code modifications, and the identification of the appropriate locations for fault injection is dependent on the exact fault type (see [12] for more details). Moreover, we are specifically interested in the faults that are representative of actual residual faults discovered in deployed systems.

We selected the fault types to inject in our experiments based on information obtained on field data from previous works (see Table 1). These fault types represent the most frequent fault types observed in the field study [12] and cover about 50% of the 532 faults analyzed in the field study, belonging to four different ODC classes. The percentage shown in Table 1 (third column) is the best approximation for the distribution of these types of faults in the code and was observed in a field study.

The definition of the set of faults to inject is based on a simple algorithm: taking into account the fault types presented in Table 1, we analyze the target code and identify all locations were a given fault can be realistically emulated (by realistically emulated we mean that the intended fault could indeed be present at the original source-code construct relative to that location in the target executable code).

We identified 232 faults for the CDMS/RTEMS setup (recall that the targets for the injection of fault in the code are the components used by the mission scenario, not all components in the system). Concerning the Ozone/007, we observed that the interface between the OO7 and the Ozone is done primarily through the OO7 class *BenchmarkImpl*. In this case, the code of all the methods of this class provided 77 faults.

Each fault is injected separately from the others. Additionally, each fault is present from the beginning of the experiment to its end. This is in accordance to the notion that a software fault is a permanent fault (i.e., it is not a transient fault). Thus, each fault implies a completely new experiment (involving the execution of the entire workload).

4.2. Set of interface faults

Concerning the injection of interface faults, the most relevant issues are the selection of the relevant interface (API), and the definition of the values to inject in the parameters of the selected interface API. Recall that the injection of interface faults consists in corrupting the parameter values of the API/methods.

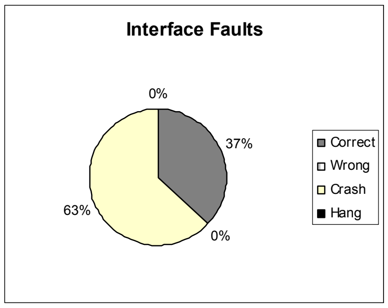
We followed the typical methodology used in robustness testing, which recommends the injection of extreme values for the parameter data type (e.g. for an integer value, 4294967295), specific values typically associated to well defined meanings (e.g., NULL, 0, -1, etc.). We also injected some intermediary values with increasingly large intervals (e.g., 10, 100, 1000, etc.). Normally extreme values of the data type are used to verify the efficiency of exception handling mechanisms and valid/invalid inputs are used to corrupt data going from one component to its successor component in order to simulate failures of the predecessor component [31]. We choose these values to observe their impact (as the use of all possible

values is impracticable) to verify if the same experiments used in robustness testing can be useful for general approach experiments.
The injection of the interface fault is a straightforward process: a list of API/methods and the corresponding parameters and values to be injected, in accordance with each data type, is provided to the fault injector (Jaca or Xception, depending on the experimental setup) and most of the process is automatic.

The injection of each interface fault corresponds to the execution of a complete experiment involving the execution of the entire workload. In other words: during one experiment only one value is used in one parameter on one API/method. Note that the fault injection can occur many times during one experiment (one for each time the API/method is invoked).
Regarding the CDMS/RTEMS setup, we defined a total of 3384 different faults. Concerning the Ozone/OO7 setup, we observed that the Ozone classes used in the interaction between Ozone and OO7 are only the *RemoteDatabase* class. The methods of this class that are public and actually implemented in the class (and not in a base class) are only three (*open*, *createObject*, and *objectForName*). Thus, the number of interface faults is less than in the other setup, consisting in 46 faults only, which is not sufficient to provide fully statistically significant results. Nevertheless, we decided to include the experiments with the Ozone and OO7 setup, as they are useful as a first insight and represent a completely different scenario when compared to CDMS/RTEMS setup.

4.3. Experimental results

We defined four failure modes for the Ozone/OO7 setup: Hang (the OO7 benchmark does not terminate in the allotted time), Crash (the OO7 terminates abruptly before completing the assigned workload), Wrong (the workload terminates but the resulting information in the database is not correct when compared to the execution without faults), and Correct (there are no errors reported and the resulting information is correct).
Figure 4 shoes the results obtained in the Ozone/OO7 setup. We observed that the behavior of the system when subjected to faults in the target code differs substantially from the behavior when subjected to interface faults. Interface faults caused a greater number of failure occurrences and the failure mode pattern is completely different form the one observed



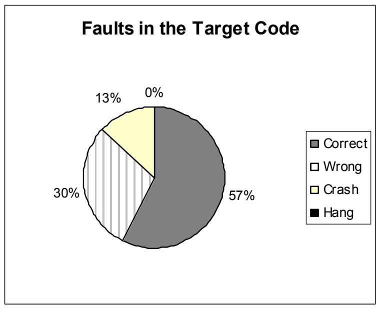


Figure 4: Interface Fault Injection Results

with faults injected in the code of components. Considering that the faults injected in the target code are representative of residual software defects (according to our previous field data works), the results suggest that interface faults are not representative of realistic software defects in the target code (at least for this setup).

Given these results, we analyzed more closely the log files produced during the experiments to better understand how the errors are propagated from the target (OO7 class *BenchmarkImpl*) to Ozone. More precisely, we wanted to discover if there is a specific set of methods and parameter values related to the situations where the behavior of the system was not correct. If indeed there is an identifiable set of methods and parameter values, then this information is relevant to guide the robustness testing experiments.

We compared the log files that describe the sequence of all methods invoked by the OO7 when no faults were injected with the files produced when a fault was injected in the code (each injected fault has its own log file). This comparison allows us to identify any differences concerning different methods being called and different values being used in the parameters. We observed that the methods being called are the same, but at some point the values used in the parameters differ. These differences are related to a parameter of the data type *reference,* and the wrong value is not a NULL value or any other remarkable or identifiable value. Additionally, these differences tend to appear in two methods (*readExternal* and *writeExternal*). This information allows the refinement of the interface faults experiments towards the improvement of injected interface faults in order to approximate as close as possible from software fault impact. In this case, faults should be centered in the *reference* data type.

We analyzed the contribution to the failure modes from each specific value used in the injection of interface faults in the Ozone/OO7 setup. These values range from -2^{31} to 2^{31}. The result of this analysis is presented in Table 2 in the form of percentage. The results show that each value has approximately the same contribution to the overall failure mode distribution. This result is surprising as it suggests that the value injected at the interface is not relevant to the system behavior. Thus we can infer that the system behavior is more dependent on the internal system architecture and implementation than in the values injected in the interface, and therefore it seems that interface faults do not represent residual software faults.

Table 2 – Contribution of each value injected at the interface to the overall failure mode (Ozone/OO7 setup)

	-2^{31}	-100	-1	0	1	10	100	1000	2^{31}	String	Null	All
Correct	50	50	50	50	50	50	50	50	50	29	29	37
Wrong	0	0	0	0	0	0	0	0	0	0	0	0
Crash	50	50	50	50	50	50	50	50	50	71	71	63
Hang	0	0	0	0	0	0	0	0	0	0	0	0

Table 3 – Contribution to the overall failure mode from each value injected at the interface (CDMS/RTEMS setup)

	0	1	64	65	4K	4K+1	256K	256K+1	16M	16M+1	2^{31}	$2^{31}+1$	$2^{32}-1$	Null	All
Correct	43	46	50	50	34	33	35	36	32	33	42	43	41	27	39
Wrong	0	1	0	0	0	1	1	1	1	1	1	1	1	7	0
Crash	48	40	39	38	52	52	57	54	61	58	51	47	52	55	50
Hang	9	13	11	12	14	14	7	9	6	7	7	9	6	11	9

The four failure modes for the CDMS/RTEMS setup are the same defined previously: Hang, Crash, Wrong, and Correct, with the same meaning as used before. The results obtained in the CDMS/RTEMS setup are those presented in the charts of Figure 5, where some of the injected module is compared (first row presents the results of the injection through the interface and second row the results of the injection into the classes).

As in the Ozone/OO7, the results suggest that faults injected at interface are not representative of faults in the component code even when we compare the results obtained for each class. The current Xception configuration did not allowed us to trace all the API calls as in the previous setup; therefore we cannot easily identify the set of API calls, parameters and values responsible for the error propagation when a fault is injected in the target component code.

Analyzing the contribution to the failure modes using the values that were injected in this setup for interface faults as well we obtained the results presented in Table 3 in the form of percentage. The results also show that each value has approximately the same contribution to the overall failure mode distribution (this finding is in accordance to the results of the previous setup and suggests that the values used in the injection at interface level are not correlated to the resulting failure modes).

Although more experiments and more setups are required to fully validate this claim, these results support the notion that, at least for some classes of

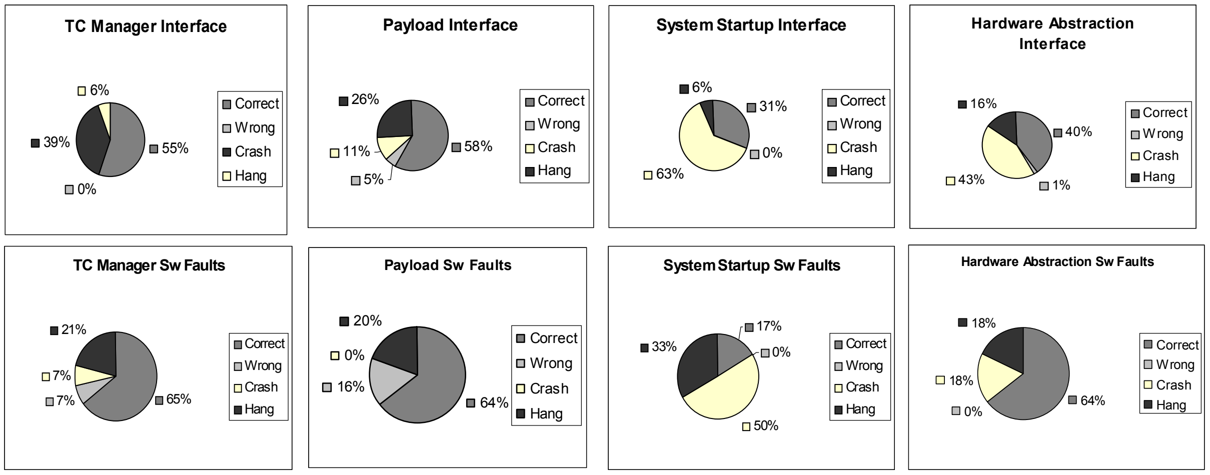


Figure 5 – Results obtained in the CDMS/RTEMS setup

systems, interface faults do not represent well software faults in component code. This means that the injection of faults at component interfaces and at the component code level should be regarded as complementary techniques, and one technique cannot be replaced by the other.

In practice, we would like to use interface faults instead of the more complex injection of faults in the code of the component, but the results obtained in our experiments do not recommend that.

5. Conclusion

This paper presents an experimental assessment of the equivalence of faults injected in the interface of components and faults injected in the code of the components. The goal was to compare the results obtained with the two fault injection techniques in order to draw a possible equivalence between internal component faults and interface faults. As component interface faults are (in general) much easier to inject than faults in the component code (as interface fault injection uses the regular component interfaces), the establishment of a possible equivalence between the two types of faults would be very useful to simplify and speedup experiments (by replacing the complex injection of faults in the component code by interface faults) or to devise coordinated ways to use both injection techniques.

The experiments were performed in two quite different setups in order to cover a good variety of scenarios. It includes a database benchmark running on top of an object-oriented database written in Java and a real-time space application written in C and running on top of a well known real-time operating system (the RTEMS). As fault injection tools we used the G-SWFIT for the injection of software faults in the code of components and Xception and Jaca for the injection of interface faults (in the C and Java environments, respectively).

The observed results suggest that interface faults do not represent well residual software faults, as the behavior of the system when interface faults are injected differs substantially from the behavior observed when faults are injected in the code of components. This means that we cannot replace the injection of faults in the code of components by the injection of faults at the component interfaces, which is normally much simpler. On the contrary, our results suggest that interface faults and faults injected in the component code are complementary techniques and one cannot replace the other.

Another interesting conclusion is that the values used in the injection of interface faults are not particularly relevant to determine the impact produced in the target system. On one hand, this conclusion also supports the non-equivalence between interface faults and faults in the code of the preceding component; on the other hand, this fact can be used to reduce the number of values used in the experiments using interface fault injection, as a big variety of erroneous input values does not seem to introduce visible benefits in the experiments.

Future work includes new experiments with more programs and the improvement of the tracing abilities of the tools in order to better identify the relationship between faults in the component code and error propagation-paths.

Acknowledges

The authors thank CAPES (Brazil), FAPESP (Brazil), GRICES (Portugal) and FCT (Portugal) to partially support this work.

6. References

[1] Arlat, J., Crouzet, Y. “Faultload Representativeness for Dependability Benchmarking”. Workshop on Dependability Benchmarking, DSN02, 2002.

[2] Bieman, J., Dreilinger, D., Lin, L. "Using Fault Injection to Increase Test Coverage". Proc of The 7th IEEE International Symposyum on Software Reliability Engineering, ISSRE'96, New York, NY, USA, 1996.

[3] Blough, D., Torii, T. "Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message Passing Parallel Computers". Proc. of The 27th IEEE Int.Fault Tolerant Computer Symposium, FCTS-27, Seattle, USA, 1997, pp. 258-267

[4] Carey, M. J. DeWitt, D. J. Naughton, J. F. “The OO7 Benchmark” http://www.columbia.edu/, 1994, accessed Feb/2006.

[5] Carreira, J., Madeira, H., Silva, J. "Xception: Software Fault Injection and Monitorintg in Processor Functional Units". IEEE Trans. on Software Engineering, vol. 24, 1998.

[6] Chiba, Shigeru. “Javassist – A Reflection-based ProgrammingWizard for Java”, proceedings of the ACM OOPSLA’98 Workshop onReflective Programming in C++ and Java, Oct. 1998.

[7] Chillarege, R. "Orthogonal Defect Classification". Handbook of Software Reliability Engineering, M. Lyu, Ed.: IEEE Computer Society Press, McGraw-Hill, Ch. 9, 1995.

[8] Christmansson, J., Chillarege, R. “Generation of an Error Set that Emulates Software Faults”. Proc. of The 26th IEEE Fault Tolerant Computing Symp. – FCTS-26, Sendai, Japan, 1996.

[9] Christmansson, J., Hiller, M., Rimén, M. “An Experimental Comparison of Fault and Error Injection”. Proc. of The 9th Int.Symposium on Software Reliability Engineering – ISSRE 98, pp. 369-378, 1998.

[10] Dingman, C., Marshall, J., Siewiorek, D. "Measuring Robustness of a Fault Tolerant Aerospace System". Proc. of The 25th IEEE International Symp. on Fault Tolerant Computing - FTCS'95, Passadena, pp. 522-527, CA, USA, 1995.

[11] Durães, J., Madeira, H. “Emulation of Software Faults by Educated Mutations at Machine-Code Level”. Proc. of The Thirteenth International Symposium on Software Reliability Engineering – ISSRE’02, Annapolis, USA, 2002.

[12] Durães, J., Madeira, H. “Definition of Software Fault Emulation Operators: A Field Data Study”. Proc. of The International Conference on Dependable Systems and Networks – DSN2003, pp. 105-114, San Francisco, USA, 2003.

[13] Fabre, J.-C., Salles, F., Moreno, M., Arlat, J. "Assessment of COTS Microkernels by Fault Injection". Proc.of The 7th IFIP Working Conference on Dependable Computing for Critical Applications- DCCA'99, pp. 25-44, San Jose, CA, USA, 1999.

[14] Fabre, J. C., Rodríguez, M., Arlat, J., Salles, F., Sizun, J. "Bulding Dependable COTS Microkernel-based Systems using MAFALDA". Proc. of the 2000 Pacific Rim International Symposium on Dependable Computing - PRDC'00, pp. 85-92, 2000.

[15] Ghosh, A., Schmid, M., Shah, V. "Testing the Robustness of Windows NT Software". Proc. of the 9th IEEE International Symposium on Software Reliability Engineering - ISSRE'98, pp. 231-236, 1998.

[16] Ghosh, A., Shah, V., Schmid, M. "An Approach for Analyzing the Robustness of Windows NT Software".Proc.of The 10th IEEE International Symposium on Software Reliability Engineering - ISSRE'99, 1999.

[17] Hiller, M., Jhumka A., Suri , N. “An Approach for Analysing the Propagation of Data Errors in Software”. Int. Conf. on Dependable Systems and Networks, DSN, Gothenburg, Sweden, 2001.

[18] Koopman, P., Sung, J., Dingman, C., Siewiorek, D. Marz, T. "Comparing Operating Systems Using Robustness Benchmarks".Proc.of The 16th International Symposium on Reliable Distributed Systems - SRDS'97, Durham, NC, USA, pp. 72-79, 1997.

[19] Koopman P., DeVale, J. "The Exception Handling Effectiveness of POSIX Operating Systems" IEEE Transactions on Software Engineering, vol. 26, pp. 837-848, 2000.

[20] Koopman, P. “What’s Wrong With Fault Injection As A Benchmarking Tool?”, in Proc. of The Internat. Conf. on Dependable Systems and Networks – DSN2002, Washington D.C, USA, 2002.

[21] Kropp, N., Koopman, P. & Siewiorek, D., “Automated Robustness Testing of Off-the-Shelf Software Components,” 28th Fault Tolerant Computing Symposium, pp. 230-239, 1998.

[22] Madeira, H. Vieira, M., Costa, D. “On the Emulation of Software Faults by Software Fault Injection.”. Proc. of The Int. Conf. on Dependable System and Networks – DSN00, NY, USA. (2000).

[23] Martins, E.; Rubira, C. M. F.; Leme N.G.M. “Jaca: A reflective fault injection tool based on patterns” Proc of the 2002 Intern Conference on Dependable Systems & Networks, pp. 483-487, Washington D.C. USA, 23-267, 2002.

[24] Moraes, R; Martins, E “A Strategy for Validating an ODBMSComponent Using a High-Level Software Fault Injection Tool”, proc.of the First Latin-American Symp, pp. 56-68, SP, Brazil, 2003.

[25] Moraes, R. and Martins, E. “An Architecture-based Strategy for Interface Fault Injection”, Workshop on Architecting Dependable Systems, IEEE/IFIP International Conf. on Dependable Systems and Networks, Florence, Italy, June 28 – July 1, 2004.

[26] Mukherjee A.,Siewiorek, D. "Measuring Software Dependability by Robustness Benchmarking," IEEE Transactions on Software Engineering, vol. 23, 1997, pp. 366-378.

[27] Ng, W., Aycock, C., Chen, P. "Comparing Disk and Memory's Resistance to Operating System Crashes". Proc. of The 7th IEEE International Symposium on Software Reliability Engineering, ISSRE'96, New York, NY, USA, 1996.

[28] Ng W., Chen, P. “Systematic improvement of fault tolerance in the RIO file cache”. Proc. of The 30th IEEE Fault Tolerant Computing Symp., FTCS-29, Madison, WI, USA, 1999.

[29] T. Tsai and N. Singh, "Reliability Testing of Applications on Windows NT", in Proceedings of the IEEE International Symposium on Dependable Systems and Networks - DSN'00, New York, NY, USA, pp. 427-436, 2000.

[30] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting How Badly 'Good' Software can Behave", IEEE Software, 1997.

[31] J. Voas, “A Defensive Approach to Certifying COTS Software”, Reliable Software Technologies, (1997).

[32] Weyuker, E.J. “Testing Component-Based Software: A Cautionary Tale”. IEEE Software, pp. 54-59, 1998.