Xception fault injection and robustness testing framework: a case-study of testing RTEMS

R. Maia[1], L. Henriques[1], R. Barbosa[1], D. Costa[1], H. Madeira[2]

[1]Critical Software SA
Parque Industrial de Taveiro, Lote 48, 3045-504 Coimbra, Portugal

[2]DEI-CISUC, University of Coimbra
Polo II, 3030-199 Coimbra, Portugal

{rmaia, lhenriques, rbarbosa, dino}@criticalsoftware.com,
henrique@dei.uc.pt

Abstract. *Xception is an automated and comprehensive fault injection and robustness testing environment that enables accurate and flexible V&V (verification & validation) and evaluation of mission and business critical computer systems and computer components, with particular emphasis to software components. In this paper we focus on the new robustness testing features of Xception and illustrate them with a concrete example of robustness testing of the Real Time Executive for Multiprocessor Systems (RTEMS) performed under a European Space Agency (ESA) contract. To the best of our knowledge, this is the first time that robustness testing results for this real time operating system are presented. The testing revealed a significant number of critical flaws in RTEMS 4.5.0 and shows the effectiveness of Xception toolset.*

1. Introduction

Test and dependability evaluation of computer systems and components are complex tasks and the growing complexity of both the hardware and software tend to make them even more difficult. Even the e valuation of very specific mechanisms such as error detection and recovery methods, which are relatively minor parts of the overall dependability evaluation, is traditionally a challenging task. Furthermore, the notion that functionality and performance are related to dependability is an established facet, as many systems operate often in degraded modes or with reduced performance due to faults.

Given the complexity of the task, there is no single generalized approach for testing and evaluating dependability features. Instead, several methods have been used ranging from pure modeling and analytical techniques to simulation and experimental approaches based on fault injection and robustness testing. These methods essentially complement each other, and depending on the phase of the design process, some of these methods could be more adequate or easy to use than others.

In general, analytical modeling and simulation are used to support architectural decisions at design phase. Detailed modeling techniques are also used for the evaluation of the dependability measures of the computer prototypes or even computer systems in field operation, but in these cases modeling needs the support of experimental approaches such as fault injection and field measurement. The experimental techniques

are used in computer prototypes or actual systems for system/components verification and validation, and constitute a very effective way to assess the efficiency of fault-tolerance mechanisms and to obtain the detailed characterization of the behavior of the whole system (or specific components) in the presence of faults or stressful conditions.

Xception is a comprehensive set of tools for experimental dependability evaluation. The Xception family started ten years ago with the proposal of a very low intrusiveness SWIF (software implemented fault injection) tool specifically targeted for very complex processors [Madeira 95, Carreira 95]. Since then, Xception has evolved to a comprehensive framework for experimental test and evaluation for dependability mechanisms and features, including several fault injection methods for a variety of target systems and a complete set of modules to assist the execution of testing campaigns and analysis of results. Xception has been used in many research and industrial projects and is being marketed by Critical Software SA (www.criticalsoftware.com).

In this paper we present the new robustness testing features of Xception and illustrated these features with an actual case-study of robustness testing of the RTEMS. Next section briefly describes Xception framework. Section 3 focus the robustness testing features and section 4 presents detailed example of the type of robustness tests that can be done by Xception. Section 5 concludes the papers and briefly describes future features.

2. Xception overview

2.1. Hybrid SWIFI Xception core

Xception has started and is best known as a SWIFI tool [Carreira 98]. The basic idea of SWIFI consists of interrupting the application under execution in some way and executing specific fault injection code that emulates hardware faults by inserting errors in different parts of the system such as the processor registers, the memory, or the application code. In general, the errors inserted are intended to emulate hardware transient faults, such as the ones that cause bit flips.

The first SWIFI approaches (i.e., prior to Xception proposal) inserted software traps in the code to mark the points during the code execution where the faults should be injected. However, these initial approaches had limited flexibility and the fault models were too simplistic, especially because no events related to time or data manipulation could be used as triggers. An alternative was to execute the code in trace mode, but this had enormous intrusiveness.

The Xception fault injection methodology uses a hybrid approach instead of a pure SWIFI methodology. The idea is to use the advanced debugging and performance monitoring features existing in modern processors (i.e., specific hardware inside the target processor) to inject more realistic faults by software and to monitor the activation of the faults and their impact on the target system behavior in detail. This approach has become a de facto standard for later SWIFI tools, like FTAFE [Tsai 96], MAFALDA [Rodríguez 99], and GOOFI [Aidemark 01].

The breakpoint registers available in the processors play a particularly important role in Xception basic fault injection approach, as they allow the definition of many

fault triggers, including fault triggers related to the manipulation of data. The processor is running at full speed and the injection of a fault corresponds normally to the execution of a small exception routine.

The performance monitoring hardware inside the processor is also used to collect detailed information on the behavior of the system after the injection of a fault. Particularly, the combination of the trigger mechanisms provided by the debugging hardware and the performance monitoring features of the processor, allows to monitor other aspects of the target behavior after the fault with minimal intrusion. For example, it is possible to detect if a given memory cell was accessed after the fault or if some program function (e.g., error recovery routine) was executed. Another important aspect is that, because Xception operates very close to the hardware (at the exception handler level), the injected faults can affect any process running on the target system including the kernel. It is also possible to inject faults in applications for which the source code is not available.

The hybrid SWIFI engine of Xception directly emulates physical transient faults in internal target processor units, main memory, and other peripheral devices that can be accessed by software. The fault triggers are based on breakpoint registers and allow the injection of faults in practically all circumstances related to code execution, data manipulation, or timing. The fault types consist of bit manipulations, which is a widely accepted model for hardware faults. Xception has been enhanced with specific modules to support several target processors include PowerPC, Intel Pentium and SPARC based platforms running Windows and Linux OSs and several real-time systems such as LynxOS, SMX, RTLinux, and ORK.

2.2. Current Xception toolset

The Xception tool has evolved from the initial hybrid SWIFI version into a comprehensive toolset including other fault injection methodologies and the new robustness testing features (presented in this paper for the very first time).

The Xception family includes the following tools:

- The original Xception tool, based on hybrid SWIFI technology.
- The extended Xception tool with the fault injection extensions based on scan chain technology (**SCIFI**).
- The new robustness testing tool.
- The Easy Fault Definition (**EFD**) and Xception Analysis (**Xtract**) add-on tools.

All tools share a common and key component: a graphic user interface named Experiment Manager Environment (**EME**), providing the backbone for the whole family of tools, thus offering a consistent view to the user.

The Xception toolset may result in a variety of different configurations, depending on the type of target system addressed and on the dependability evaluation needs. In addition to the generic components (EME, EFD, and Xtract), the actual Xception configuration required for each concrete case is achieved through the use of a set of modules (plug-ins) that implement specific fault injection or robustness testing methodologies for the different types of target systems. Figure 1 shows the Xception components and plug-ins currently available, including the new robustness testing plug-ins.

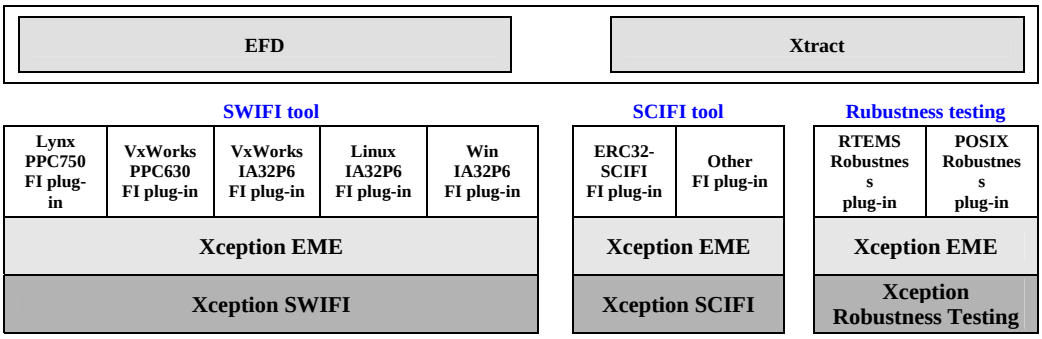

Figure 1. Xception components and plug-ins

The Xception architecture (see Figure 2) resembles the client-server model. It comprises a front-end module, which runs in a host computer and is responsible for experiment management/control (the EME components and, optionally, the EFD and Xtract components), and a lightweight injection core (FI hook) and monitoring elements (readout collector), which runs in the system under evaluation and is responsible for the insertion of the faults.

The connection between the host and the target is done by means of a high level protocol, built on top of TCP-IP. Experiment and fault configuration data flows from the host to the target, while fault injection raw data results flow in the opposite direction. The Xception tool includes all the components running on the host computer and the injection core and monitoring elements installed in the target system (see Figure 2).

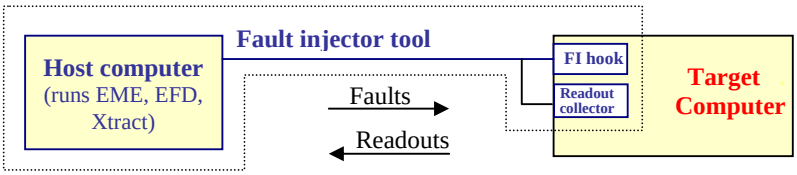

Figure 2. Xception components and plug-ins

The Xception EME (Experiment Manager Environment) running on the host computer is responsible for fault definition, experiment execution and control, outcome collection and basic result analysis. The EFD (Easy Fault Definition) is optional and is meant to facilitate the definition of faults through a high level view of the target system that allows the user to define faults triggers pointing directly to target code areas. The Xtract (Xception Analysis) is a stand-alone tool meant to perform detailed result evaluation, including detailed tracing of fault effects.

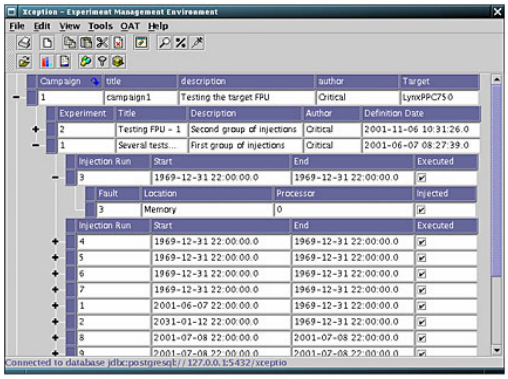

Figure 3. EME main window

Xception uses a relational database, enabling extensive outcome analysis. The Xception database holds all the information relevant to the fault injection and robustness testing process, including fault definition,

experiment execution and result analysis. The use of an open standard such as SQL (Structured Query Language) also allows the user to execute specific SQL queries or use other tools available in the market to explore the results, in addition to the tool already provided by the Xception.

3. Xception robustness testing methodology

The robustness testing methodology used by Xception is inspired on the robustness testing proposals from Carnegie-Mellon University [Koopman 97, Koopman 99] and LAAS [Rodríguez 99, Arlat 02]. The idea of classical robustness testing is to evaluate API (application programming interfaces) robustness in the presence of invalid inputs parameters. This idea has been applied to operating system calls but it can be generalized to inject faults in the interface parameters of any software component (we actually did that generalization in Xception robustness testing).

In robustness testing, interface faults change the input parameters turning them into invalid or exceptional values that may cause component failures. In principle, software components should behave in a robust way, even when they are submitted to invalid parameters (for example returning an error code). Unfortunately, most of the APIs are not robust, either because engineers assume that it is not practical to test and handle all the possible invalid inputs or because they just decide to overlook some input tests for performance reasons. The reality is that many API's are not robust as has been shown in several robustness testing works (e.g., [Koopman 99, Rodríguez 99]).

Once the input patterns that cause a given component to fail are identified, the problem can be solved either by changing the API, if the source code is available, or using wrapping when the source code is not available, which is normally the case of COTS components.

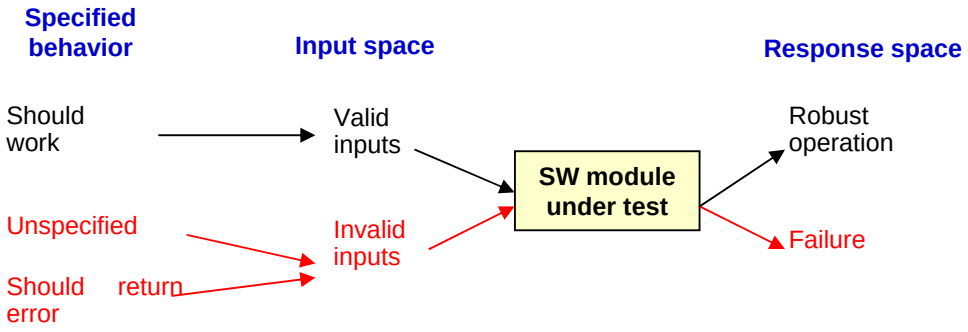

Figure 4 - Classical ([Koopman 99]) robustness testing approach.

Compared to classical approaches, Xception robustness testing approach includes some new aspects:

- It allows the inclusion of specific environment requirements (i.e., requirements imposed by other components of the target system or by the requirements external) in the robustness test cases. This includes, for example, timing requirements such as minimum response time in the presence of invalid input parameters.

- It is not specifically tied to operating systems or micro-kernel APIs (application programming interfaces) as it can be used to test the robustness of interfaces of any software component, including COTS (commercial off-the-shelf) and custom components. This generalization of robustness testing approaches is similar to Interface Propagation Analysis [Voas 98] and to the technique proposed in [Moraes 04].

  The Xception robustness testing methodology comprises three phases (see Figure 5):

- **Preparation**: Includes all the tasks needed to define test cases.
- **Test Execution**: Execution of the defined test cases.
- **Log Analysis**: Analysis of the results of the test cases and identification of the RTEMS faults.

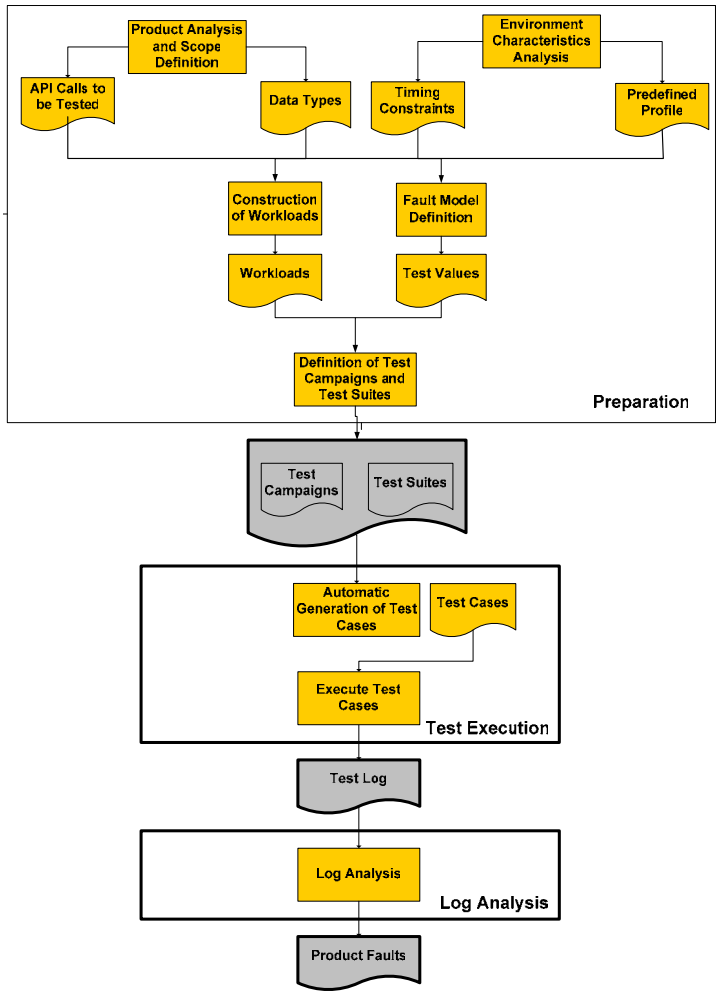

Figure 5 - Xception robustness testing methodology.

Preparation phase comprises the following tasks:

- Product Analysis and Scope Definition: Analysis of the product/component under evaluation and selection of the API calls that will be subjected to the evaluation.
- Environment Characteristics Analysis: Identification of extra requirements imposed by the environments (other target components and external environment) in the product/component under evaluation.
- Fault Model Definition: Definition of the in-bound and out-of-bound values that will be used as interface faults for each data type of the product/component under evaluation.
- Construction of the Workloads: Definition and implementation of the applications that will exercise the product/component under evaluation APIs.
- Definition of the Test Campaigns and Test Suites: Definition of the test suites that will be used to automatically generate the test cases (using the workload and fault model previously defined). Test suites are grouped logically in test campaigns.

The Test Execution phase follows the Preparation. During this phase test cases are executed and the results are collected to the Xception database. This task is performed in unattended mode by Xception.

The final phase of the robustness testing is the Log Analysis. In this phase detailed analysis of the log of each test case is performed comparing the obtained results against the expected values. This phase may be quite time consuming. For this reason it is important to have a concise workload output that enables the analyst to quickly find out if the result of the test case is consistent with the input parameters or not.

A very important aspect of robustness testing is the definition of the **interface faults** for each data type (see preparation phase described above). For each data type, a class of test values is defined in order to facilitate the definition of the test cases. For practical reasons, a test case is generated as mutants of the workload. Each mutant is the source code file of the workload that results from the application of a single mutation on a data parameter. These values differ from typical data values fed to functional tests (or unit tests) since they are specifically meant to exercise the error handling and robustness features of the component under test. These values typically reflect boundary or "magic" values in the data type range, or values that are semantic out-of-bounds in the scope of usage in a function call.

The test values (interface faults) for the basic data types were defined taking into account preparatory experiments and published data on robustness testing techniques, namely from Ballista project [Koopman 99]. The test values candidates are as follows (assuming C syntax):

- **Integers data types**: 0, 1, -1, MAX_INTEGER, MIN_INTEGER, selected powers of two, powers of two minus one, powers of two plus one.
- **Pointers data types**: NULL, -1 (cast to a pointer), pointers to free()'ed memory, pointers to malloc()'ed buffers of various powers of two in size.
- **Floats data types**: 0.0, 1.0, -1.0, ±MAX_FLOAT, ±MIN_FLOAT, PI and e.

In order to prevent an explosion in the number of test cases, but keeping good test coverage at the same time, a subset of the specified test values are normally selected. These test values do not represent all the possible values that may be interesting to test with the given data types. They have been chosen to provide a reasonable range of exceptional (non-nominal) input conditions to the software under test. Still, some rules apply. If the type to be used in a mutation is a pointer to a function, then the only possible value that will be used to create mutants is the NULL pointer. This decision is based on the fact that it is not interesting to pass invalid function pointers as parameters.

A final aspect (and also very important) is the failure mode classification. That is, when injecting interface faults in a given software component it may fail in several ways (failure modes). Xception has adopted the Ballista classification [Kropp 98] as a first step classification scale. This scale is known as CRASH and measures non-robust responses from the component under test. CRASH is an acronym for the following failures modes:

- **C**atastrophic failures: a complete system crash that requires a system reboot.
- **R**estart failures: the application hangs and requires application restart.
- **A**bort failures: abnormal termination of an application.
- **S**ilent failures: occur when it is reported that the called function or system call was completed successfully instead of returning an error indication (as it would be expected due to the invalid parameter values used in the call).
- **H**indering failures: incorrect error indication such as the wrong error reporting code.

As mentioned, Xception uses the CRASH scale only as a first step classification. In fact, as the impact of a robustness failure is highly dependent on the concrete application, in many cases is enough to identify robustness failures, without going into a detailed classification that may not apply to the concrete testing scenario. Another limitation of the CRASH scale is that it is assumes robustness testing of operating systems and is not adapted to other software components.

The natural alternative to the CRASH scale is to use the simplified scale suggested by Figure 4, which consider two possible responses: **robust operation** and **robustness failure**. In many cases it is useful (and easier) to classify robustness failures in critical and minor failures, according to the fault impact.

4. RTEMS case-study

This section presents a robustness testing study of the RTEMS 4.5.0 (Real Time Executive for Multiprocessor Systems) performed under a European Space Agency (ESA) contract.

4.1. RTEMS and experimental setup

Figure 6 shows the application architecture of RTEMS. It includes three sets of APIs: classic, POSIX and Itron. Only the first two API sets were tested in the present study.

The internal architecture for RTEMS can be regarded as a set of layers that work closely with each other to provide the necessary services to real time applications. The executive interface presented to the application is formed by grouping *directives* (API calls) into logical sets called resource managers. Scheduling, dispatching and object management are provided by the executive core, which depends only on a small set of CPU dependent routines.

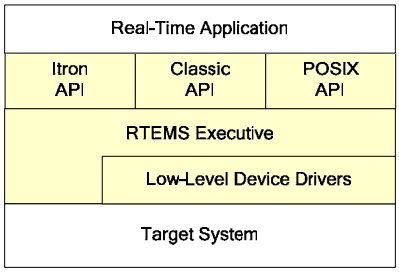

Figure 6. RTEMS Application Architecture

The Classic API provides seventeen resource managers as presented in Figure 7. For space reasons, we do not describe the resource managers as they represent traditional features provided by operating systems to real-time applications (a detailed description can be found in www.rtems.com).

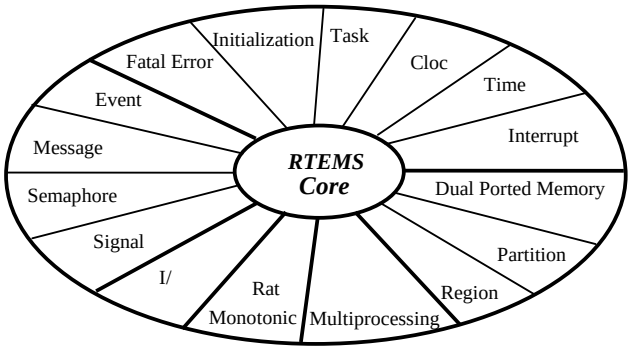

Figure 7. RTEMS Classic API resource managers

The POSIX API is based on the standard IEEE 1003.1b and provides nineteen managers, as shown in Figure 8.

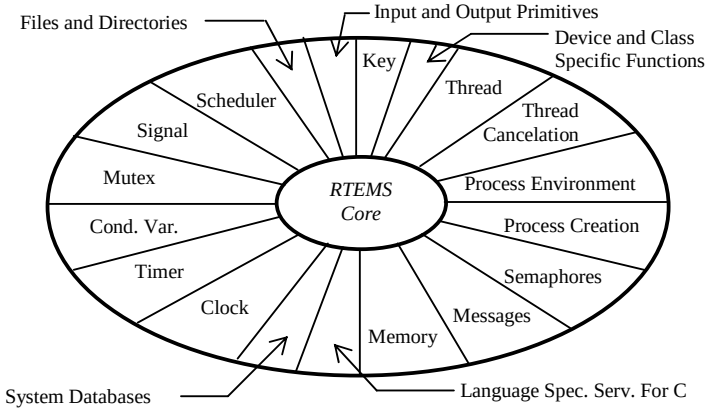

Figure 8. RTEMS POSIX API resource managers

The RTEMS executive core is responsible for the low level system management, including all CPU specific features and low level device drivers (see Figure 6). In the version tested, the CPU specific features belong to the SPARC/ERC32 processor. The RTEMS executive core also implements several handlers to be used by the API's to perform each specific function. These handlers include the message handler, the mutex handler, the semaphore handler, etc. The core was not designed to be used directly by the user's application, although no restrictions are imposed, at either compilation or linking time.

The experimental setup is the typical Xception setup, including the host computer and the target system, where the RTEMS is running (see Figure 9). For practical reasons, we have used a target system simulator instead of an actual board with the ERC32 processor. However, as the code executed in the target simulator is the actual RTEMS 4.5.0 code, the results are not affected by the fact the RTEMS is running in a processor simulator instead of an actual ERC32 board.

The Xception host is responsible for all the phases of the robustness testing experiment, as described in section 2. During the execution of the test cases, Xception creates the mutants following the test case definition. Then, the executable binary is built and the faulty application (mutant) is finally uploaded to the target system and executed. Data related to the execution is logged in order to allow the result analysis process.

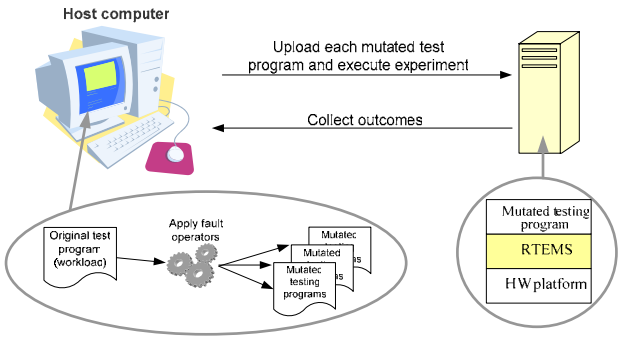

Figure 9. Experimental setup

4.2. Results

Tables 1 and 2 shows the summary of the results obtained for the resource managers already tested for the Classic API and the POSIX API, respectively. The first thing to note is the high number of robustness failures found in RTEMS 4.5.0, especially to what concerns critical failures.

We used the simplified classification of robustness failures mentioned in section 3 (instead of the CRASH classification). All the robustness failure situations have been manually analyzed and classified as **critical** or as **minor** failures, depending on the complexity of the recovery that would have been required in a real situation. In general critical failures include system and application crashes and minor failures include hindering cases. In most of the minor failures, RTEMS has returned a wrong error code.

The detailed analysis of the robustness failures cases has shown a variety of situations, all of them representing critical weak points of RTEMS. Note that all the failure cases

represent situations in which the RTEMS should have behaved in a robust way, for example returning a correct error code to the test program (instead of crashing or producing other unacceptable behavior). The following points show some examples of erroneous behavior observed:

- Unexpected change of the program control flow (e.g., when the task identifier is set to 0 on the rtems_task_start call).
- In several situations the test cases end up with unhandled traps such as *data access*, *memory not aligned*, and *illegal instruction* exceptions.
- Several situations where the RTEMS did not return any error code (silent failures). These cases are particularly dangerous, as they may affect the application in an unpredictable way.
- Several hindering failures where wrong error codes have been returned (i.e., instead of returning the error code specified in the documentation a different error code is returned).

Table 1. Classic API robustness testing results

Manager	Test Cases	Robustness failures		
		Critical	Minor	Total
Clock	68	0	0	0
Event	18	0	0	0
Fatal Error	3	0	0	0
Interrupt	5	0	0	0
IO	50	5	1	6
Message	83	2	6	8
Partition	27	0	2	2
Rate Monotonic	24	0	1	1
Region	67	4	3	7
Semaphore	33	0	1	1
Signal	10	0	1	1
Task	55	2	2	4
Timer	67	2	1	3
User Extensions	17	0	1	1
Total	**527**	**15**	**19**	**34**

Table 2. POSIX API robustness testing results

Manager	Test Cases	Robustness failures		
		Critical	Minor	Total
Clock	32	0	0	0
MESSAGE	122	2	1	3
MUTEX	223	1	3	4
Signal	122	1	4	5
Timer	29	0	3	3
Total	**528**	**4**	**11**	**15**

Due to space restrictions we cannot present a more detailed description of the results observed. The interested reader can find the full results in [Maia 04]. The detailed results also include code coverage results showing the lines of robustness testing code executed in each test case (zero lines in many of them) and the results of a comprehensive stress testing campaign (i.e., workloads that make extremely high usage of system resources).

4. Conclusion

This paper surveys the current features of the Xception fault injection and robustness testing family of tools. The main focus is on the new robustness testing features of Xception that are illustrated through a concrete case-study of testing the robustness of the RTEMS 4.5.0. To the best of our knowledge, this is the first time that robustness testing results for this real time operating system are presented. The results show a significant number of robustness failures situations (49 failures in total), including a large percentage of critical failures. These results clearly show that RTEMS should be improved for robustness before being used in critical applications.

References

Aidemark J., Vinter J., Folkesson P., Karlsson J., "GOOFI: Generic Object-Oriented Fault Injection Tool", Proceedings of Int. Conf. on Dependable Systems and Networks, DSN-2001, Göteborg, Sweden, 2001, pp. 71-76.

Arlat J., Fabre J.-C., Rodríguez M., and Salles F., "Dependability of COTS Microkernel-based Systems", IEEE Trans. on Computers, 51 (2) February 2002, pp 138-163.

Carreira J., Madeira H., and Silva J. G., "Xception: Software Fault Injection and Monitoring in Processor Functional Units", 5th IFIP Working Conference on Dependable Computing for Critical Applications, DCCA-5, Urbana-Champaign, Illinois, USA, September 27-29, 1995.

Carreira J., Madeira H., and Silva J. G., "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", IEEE Trans. on Software Engineering, 24 (2), pp.125-36, February 1998.

Koopman P. and DeVale J., "Comparing the Robustness of POSIX Operating Systems", in Proc. 29th Int. Symposium on Fault-Tolerant Computing (FTCS-29), (Madison, WI, USA), pp.30-7, IEEE CS Press, 1999.

Koopman P., Sung J., Dingman C., Siewiorek D., Marz T., "Comparing Operating Systems using Robustness Benchmarks", in Proceedings of the 16th International Symposium on Reliable Distributed Systems, SRDS-16, Durham, NC, USA, 1997.

Kropp, N., Koopman, P., and Siewiorek, D., "Automated Robustness Testing of Off-the_Shelf Software Components," 28th Fault Tolerant Computing Symposium, June 23-25, 1998.

Madeira H., Carreira J., and Silva J. G., "Injection of faults in complex computers", Fourth IEEE International Workshop on Evaluation Techniques for Dependable Systems, San Antonio, Texas, USA, October 2-3, 1995.

Maia R. et al, "RTEMS 4.5.0 Evaluation Report", CSW-RAMS-2003-CTR-1306, RAMS Call-off Order 2, ESTEC/Contract Nº 16582/02/NL/PA, 2004 (available on specific request to rmaia@criticalsoftware.com).

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.

Rodríguez M., Salles F., Fabre J.-C., and Arlat J., "MAFALDA: Microkernel Assessment by Fault Injection and Design Aid", in *Proc. 3rd European Dependable Computing Conf. (EDCC-3)*, (E. M. J. Hlavicka, A. Pataricza, Ed.), (Prague, Czech Republic), LNCS, 1667, pp.143-60, Springer, 1999.

Tsai T. and Iyer R. K., "An Approach to Benchmarking of Fault-Tolerant Commercial Systems", Proceedings of the 26th IEEE Fault Tolerant Computing Symposium, FTCS-26, Sendai, Japan, pp. 314-323, June 1996.

Voas J. and McGraw G. "Software Fault Injection: Inoculating Programs against Errors", John Wiley & Sons, New York, EUA, 1998.