FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Study and Development of a Software Implemented Fault Injection Plug-in for the Xception tool/PowerPC 750

Álvaro Manuel da Silva Monteiro

Report of Project
Master in Informatics and Computing Engineering

Supervisor: Ana Cristina Ramada Paiva Pimenta (Assistant Professor)
29 of June 2009

Study and Development of a Software Implemented Fault Injection for the Xception tool/PowerPC 750

Álvaro Manuel da Silva Monteiro

Report of Project
Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: José António Rodrigues Pereira de Faria (Assistant Professor)

External Examiner: Alberto Manuel Rodrigues da Silva (Associate Professor)
Internal Examiner: Ana Cristina Ramada Paiva Pimenta (Assistant Professor)

Abstract

System validation and evaluation of critical computer systems has become of great importance due to the relevance and value of these systems. One way to perform the dependability verification of computer systems is to resort to fault injection techniques, more precisely Software Implemented Fault Injection (SWIFI). This technique allows understanding the behaviour of a computer system by inserting artificial faults. This enables developers to create fault tolerant systems that can sustain faults in a real world environment. This project dealt with a SWIFI plug-in for the Xception tool. This project ultimate goal was to port an already existing plug-in for the PowerPC603 Microprocessor using the real-time operating system VxWorks to the PowerPC 750 Microprocessor using the same operating system. Since this tool has such a specific purpose, it was necessary to clearly define a process that could assure that the result of the port would be a proper and fully functional plug-in. To do so, a porting methodology that reflects the special needs of this kind of tool was developed. This porting methodology reflected the already defined software development process and was developed with guidelines to make the porting process more precise and seamless. In this port it was defined everything pertaining requirements, software architecture and tests as well as a Software Implemented Fault Injection methodology that allowed to contextualise scientifically the plug-in i.e., in which way this plug-in reflects the proprieties, that have been defined as essential by several studies, of a tool of this kind.

The porting methodology proved to be quite helpful as it allowed porting the plug-in successfully.

Resumo

A validação e avaliação de sistemas informáticos críticos tornou-se de grande importância devido à relevância e valor deste tipo de sistemas. Uma forma de avaliar a confiabilidade de sistemas informáticos é a de utilizar técnicas de injecção de falhas, mais precisamente, injecção de falhas por software (SWIFI). Esta técnica permite compreender o comportamento de um sistema informático com o recurso a inserção de falhas artificiais. Isto permite criar sistemas tolerantes que conseguem resistir a falhas reais aquando da sua utilização prática. Este projecto lidou com um plug-in SWIFI para a ferramenta Xception. O objectivo final deste projecto era exportar um plug-in já existente para o Microprocessador PowerPC 603e que utilizava o sistema operativo de tempo real VxWorks para o Microprocessador PowerPC 750 com o mesmo sistema operativo. Uma vez que esta ferramenta tem um propósito tão especial, foi necessário definir um processo que pudesse garantir uma exportação correcta. Para tal foi desenvolvido uma metodologia de exportação que reflectisse as necessidades especiais de uma ferramenta deste tipo. Esta metodologia reflecte o processo de desenvolvimento já existente e foi desenvolvida com linhas mestras de modo a tornar o processo de exportação mais preciso.

No início da exportação foi definido tudo relacionado com requisitos, arquitectura e testes como também uma metodologia SWIFI que permitiu contextualizar cientificamente o plug-in i.e., de que forma este plug-in reflecte as propriedades, que foram claramente definidas como essenciais em vários estudos, de uma ferramenta deste tipo.

Esta metodologia de exportação foi aplicada e resultou numa exportação com sucesso.

Acknowledgements

I would like to thank Ricardo Barbosa, my tutor at Critical Software, for his guidance throughout the project, to Ana Paiva, my tutor at FEUP for clearly defining this project, Sérgio Vidal for helping me with technical details. I would also like to thank Miguel Pinto, Ricardo Batista and Pedro Braga for their companionship and good humour.

Álvaro Manuel da Silva Monteiro

Table of Contents

List of Figures

List of Tables

Abbreviations

AR	Acceptance Review
CDR	Critical Design Review
COTS	Commercial-off-the-shelf
$\text{DBS}_{\text{specific}}$	Database Specification Specific
DBS_{tpl}	Database Specification Template
$\text{DDS}_{\text{specific}}$	Detailed Design Specification Specific
$\text{DDS}_{\text{specific-updated}}$	Detailed Design Specification Specific Updated
DDS_{tpl}	Detailed Design Specification Template
EME	Experiment Management Environment
FERARRI	Fault and ERRor Automatic Real-time Injector
$\text{FR}_{\text{specific}}$	Feasibility Report Specific
FR_{tpl}	Feasibility Report Template
HWIFI	Hardware Implemented Fault Injection
$\text{IM}_{\text{specifc-updated}}$	Installation Manual Specific Updated
$\text{IM}_{\text{specific}}$	Installation Manual Specific
IM_{tpl}	Installation Manual Template
IRC	Injection Run Controller
KOM	Kick-Off Meeting
MAFALDA-RT	Microkernel Assessment by Fault Injection AnaLysis and Design Aid of Real Time systems
MP	Maintenance Plan
$\text{MP}_{\text{specific}}$	Maintenance Plan Specific
O.S.	Operating System
PCM	Project Closedown Meeting
PDR	Preliminary Design Review
PPC	PowerPC
$\text{SAS}_{\text{specific}}$	Software Architecture Specification Specific
SAS_{tpl}	Software Architecture Specification Template
$\text{SRS}_{\text{specific}}$	Software Requirements Specification Specific
SRS_{tpl}	Software Requirements Specification Template
SWIFI	Software Implemented Fault Injection
$\text{TCS}_{\text{specific}}$	Test Case Specification Specific
TCS_{tpl}	Test Case Specification Template

$\text{TER}_{\text{specific}}$	Test Execution Report Specific
TER_{tpl}	Test Execution Report Template
$\text{UM}_{\text{specific}}$	User Manual Specific
$\text{UM}_{\text{specific-updated}}$	User Manual Specific Updated
UM_{tpl}	User Manual Template

Chapter 1

Introduction

Everything is theoretically impossible, until it is done. One could write a history of science in reverse by assembling the solemn pronouncements of highest authority about what could not be done and could never happen.

Robert Anson Heinlein

This document describes a project developed at Critical Software. The purpose of this project was to port, an already existing plug-in, and develop a fully functional plug-in for the Xception tool [CSW03]. This tool is a software implemented fault injection platform that empowers users with the ability of performing advanced and effective system validation, evaluation and testing of mission and business critical software systems through the use of a fault injection technique – Software Implemented Fault Injection (SWIFI). Fault injection is a testing technique that deals with calculated insertion of artificial faults into a computer system in order to gain an understanding of the effects of real faults and subsequently to provide with feedback for system correction and enhancement. The SWIFI technique consists in injecting faults by resorting to software.

The Xception product works with a wide variety of operating systems and hardware platforms. It uses a highly flexible architecture that allows for a constant updating (in order to meet the client's needs). The way to update this software is through the implementation of plug-ins that can perform software-implemented fault injections in one or more combinations of operating systems/hardware platform.

The ultimate goal was to analyse the existing plug-in for the PowerPC 603e Microprocessor, that works on the real-time operating system VxWorks, and port it to the PowerPC 750 Microprocessor on the same operating system. In order to do this, a port methodology was created that will ease and improve and make more seamless future ports.

1.1 Context

This project was developed at Critical Software S.A.. It is a software company which provides solutions, services and technology for critical information systems, coping with the needs of many clients from different markets, namely, telecommunications, public sector, industry and aeronautics and defense industry. The company was founded in 1998 and nowadays it employs circa 500 people in its offices in Portugal (Coimbra, Lisbon and Porto) and overseas, in the UK, USA and Romania.

The company's success relies on two ground rules – use of high quality standards and a strong investment in innovation. This management policy has led to an ever increasing portfolio of unmatched solutions that are developed on-time and on-budget. Subsequently, the number of partnerships with very demanding clients is increasing, most noticeable, the European Space Agency, National Aeronautics and Space Administration, BOEING, EADS, Japanese Space Agency among many others.

The companies mentioned above use mission and business critical systems as the cornerstone of their main activities, usually connected to aeronautics and/or space. These systems need to be tested in a very thorough manner in order to validate a whole venture. Critical Software, through its Security Critical Systems section, provides the necessary tools to verify and validate these systems.

The software application Xception provides a way for these companies to test their systems and determine how to minimize potential hazards. As mentioned before, this software uses a very specific technique to inject faults during runtime – SWIFI (Software Implemented Fault Injection). Before delving into details of the technique one should point out why it is important to validate these systems and ultimately substantiate the importance of this project.

There is a generalised trend in the critical systems industry to use commercial-off-the-shelf (COTS) components. This is motivated by the fact its use leads to a reduction in time-to-market as well as a reduction in overall development cost [Bohem99]. Nevertheless, the decision to use COTS components raises serious problems from a dependability viewpoint [Voas99].

This leads to a very simple question: how to validate systems built upon COTS components?

Before answering this question it should be explained how these systems can be affected.

Microelectronics circuits operating in space are exposed to radiation in the form of energetic charged particles, such as protons and heavy ions [Fouillat04]. This radiation can subsequently lead to malfunctioning. The short circuits provoked by these radiations can induce permanent damage by thermal effect if the device is not rapidly powered-off. Transient changes can also occur that corrupt the information stored in memory cells within integrated circuits [Velazco02].

Although the space environment is filled with hazards that can alter the normal functioning of an electronic system, similar problems can take place on a more earthly environment. Electrical and magnetic disturbances are quite common in power and signal processing electronics which are ubiquitous in all in aspects of modern life. For instance, industrial controllers, which monitor and control everything related to industrial processes, have microprocessors which can be subjected to power fluctuations and electro-magnetic radiation, which subsequently can alter the memory and register contents of a microprocessor [Caignet01].

These disturbances can occur in many different critical systems such as train control and aircraft control.

1.2 Project

The project involved three major steps: definition of a methodology for porting; definition of a software implemented fault injection methodology; porting/implementing the plug-in.

A methodology for porting the plug-in was defined, based on an analysis of the existing software development process (SDP) at Critical Software's.

The definition of a software implemented fault injection methodology comprehended a space and terrestrial domain analysis and mission profile, i.e., what are the causes of "real world" faults and its consequences for critical missions; a definition of a model, taking into account research already performed in the field, regarding the set of faults, the way they function, how to interpret the consequences and how to measure the system's dependability; a definition of how the emulation should be executed and the subsequent analysis of the results.

Implementation started when all the previous steps were completed. It involved a specific hardware and software configuration in order to deploy correctly the plug-in and careful programming for a successful port that fulfilled the defined requirements.

1.3 Motivation and Objectives

Having a plug-in that allows the Xception tool to inject faults to another microprocessor that is used in spacecrafts and aeronautics is of paramount importance. Although the PowerPC 750 Microprocessor is old, in an ever evolving technological timeframe, one must keep in mind that critical missions usually use old but tested technology and this Microprocessor will start to be used in one of NASA's venture in partnership with the Honeywell Technology Solutions. [NASA03] [McHale04]

The main objectives of this project were:

- To get acquainted with the software implemented fault injection technique.
- Conduct a state-of-the-art study about fault injection through the use of software and future developments.
- Define and use a porting methodology that will allow for future ports to be more seamless and faster.
- Define a SWIFI methodology that fits the Xception plug-in, i.e., contextualise scientifically the fault injection process developed at CSW.
- Port a fault injection plug-in for the Xception tool.

In order to achieve these objectives a modified waterfall software development approach was followed, accordingly to the Critical Software's software development process, with the purpose of clearly defining a context, requirements, architectures and a approach/methodology for the implementation. There were weekly project meetings and pre-defined milestones.

1.4 Report Overview

Chapter 1, Introduction, presentation of context of the project, its background and its motivation and objectives.

Chapter 2, State of the art, contextualizes further more this project, illustrates the research done regarding previous work and developed technologies in the field of software implemented fault injection. In this chapter it is possible to understand the several approaches there are to this technique.

Chapter 3, Xception Tool, gives detailed information about the tool developed at Critical Software using a software implemented fault injection plug-in. This description should allow a better understanding of the tool as well as its technical details.

Chapter 4, Porting Methodology, presents the defined porting methodology that was used to perform the port. It includes a description of the general software development process used at Critical Software, how this methodology relates with it and the several phases it is comprised. Each one of these phases has a set of guidelines in order to explain the procedure as well as the necessary artefacts that need to be produced.

Chapter 5, Port, consists of the solution implementation details. All relevant aspects of the system implementation are mentioned in this chapter. This includes the architecture and functions of the several components that comprehend the plug-in

Chapter 6, Conclusions and future work, it ponders the contributions of this project, the achieved results as well as future improvements.

Chapter 2

State of the Art

The beginning of knowledge is the discovery
of something we do not understand.
Frank Herbert

This chapter presents a description of fault injection technique, its purposes and variations. It presents the reasons behind the usage of SWIFI as well as related works on the subject.

2.1 Introduction

As explained in the previous chapter, it is of great importance to prevent failures of computer-driven systems since these can lead to very serious consequences regarding human lives and monetary loss. Despite the more common used methodologies to ensure the reliability, availability and safety of computer systems, such as careful and thorough design, quality assurance and other fault avoidance techniques, a proper testing procedure that ensures that a given system meets a certain level of dependability must be applied.

Before expanding more on the subject of testing procedures, one should clearly define the concept of dependability in a computer system. Dependability is the property of a computer system such that reliance can justifiably be placed on the service it delivers. The service delivered by a system it is the behaviour as it is perceptible by its user [Laprie95]. Dependability has many facets, this means it can be assessed accordingly to different points of view, although these are (must be) complementary. These faces or attributes can be defined as:

Availability: The readiness for usage [Laprie95] or more specifically the probability that a system is operating correctly and is available to perform its functions at a given instant of time [Johnson89].

Reliability: The continuity of service [Laprie95] or more specifically the probability that a system will perform in a proper way in a given interval [x0, x], assuming the system was performing correctly at time x0 [Johnson89].

Safety: The non-occurrence of catastrophic consequences on the environment [Laprie95], more specifically the probability that a system will either perform its functions correctly or will discontinue its functions in a manner that does not disrupt the operation of another system or compromise the safety of lives associated with the system [Johnson89].

Mean Time to Failure (MTTF): The expected time that a system operates before the first failure occurs [Johnson89].

Coverage: The probability of recovery of a system when a fault exists [Johnson89]. This directly connects with the ability of detection of a fault.

Maintainability: A measure of how of difficult a system can be repaired, once it has failed [Johnson89].

Testability: The possibility of determining the existence and quality of certain attributes within a system [Johnson89]. This concerns the validation and evaluation process of the system.

Please notice that when failures are mentioned it means that a fault has not been detected/handled, in order to contain possible errors that may arise, and it originates in a malfunction (failure).

Generally speaking, fault injection can be defined as a dependability validation technique that is based on controlled experiments where one can observe the system behaviour in presence of faults, which are introduced by artificial means. This technique can speed up the occurrence and the propagation of faults into the system in order to observe the effects on the system [Benso03]. It can be performed on models, working prototypes or systems in the fields [Benso03] [Arlat93].

2.2 Objectives of Fault Injection

All systems have, or should have, fault tolerance mechanisms. Fault injection tries to determine whether these mechanisms are sufficient and to what extent. Usually, the test cases, i.e., type of fault, test points, injection time and state, are designed taking into consideration a great understanding of the system at hand [Benso03].

This being said, one can group the benefits of fault injection in three main points:

- The ability to understand the effects of real faults and of the related behaviour of the system;
- Understand and assess the efficacy and efficiency of the fault tolerance mechanisms existent in the system being tested, which will allow getting a feedback to enhance and correct the system.
- Forecasting the target system's faulty behaviour, in particular encompassing a measurement of the coverage provided by the fault tolerance mechanisms.

2.3 Fault Injection Techniques

There are several ways to characterize fault injection techniques. Taking into consideration the practical scope of this project, two kinds of these techniques are considered, hardware implemented fault injection and software implemented fault injection.

2.3.1 Hardware-Implemented Fault Injections

One type of fault injection, which is quite common, is to inject physical faults into the target system hardware - hardware implemented fault injection (HWIFI). Although this method has the clear advantage of causing actual hardware faults, it requires the use of special hardware and the used tool for fault injection are, in most cases, specific to the target system [Benso99]. Generally speaking, the main advantages associated with this method are the following:

- Fast experiments.
- Can access locations that are hard by other means.
- Experiments can be run in near real-time, which allows for a great number of experiments.
- The experiments are executed using the same software that will be used in the field [Benso03].

Nevertheless, HWIFI has many disadvantages:

- The injected system is prone to damage due to the fault injection.
- Low portability.
- Limited set of injection points.
- Limited set of injectable faults.
- Complexity and speed of today’s microprocessors make the design of special hardware very difficult or even impossible.
- Need of special-purpose hardware.[Benso03]

2.3.2 Software-Implemented Fault Injections

An alternative to HWIFI technique is software implemented fault injection. It consists in interrupting the application execution and executing specific fault injection software code that emulates hardware faults in a wide variety of components, such as the microprocessor’s registers, memory or even the application code [Carreira98].

It should be pointed out that the faults injected in the system should be representative of the actual faults that can occur in the system, in the scope of the system’s purpose, and that the additional software that is necessary to inject faults must not affect the system under analysis.

This approach has many advantages:

- Can inject faults in a wide variety of systems, which can be difficult using HWIFI;
- Experiments can be run in near real-time, which allows for a great number of experiments;
- No need for special purpose hardware;
- Low implementation cost [Benso03].

- Not significantly affected by the increasing complexity of processors [Carreira98].

However, there are a set disadvantages associated with this technique. These are:

- Cannot inject faults into locations that are inaccessible to software;
- Very difficult to model permanent faults;
- The fault injection process could affect the order of the systems tasks [Benso03].

Among the advantages of SWIFI there are three that are of paramount importance: no need for special purpose hardware (which leads to a higher rate of portability), low implementation cost, and it is not significantly affected by the increasing complexity of processors. These advantages are the main reasons for the success of this technique. From now on, any commentary or description regarding fault injection should be interpreted as being applied to SWIFI, although in many cases it can be generalised.

2.4 Existing SWIFI Tools

This sub-section presents a set of existing SWIFI tools that are relevant in this domain and are comparable to the Xception tool. These tools were selected because of its importance in this field as well as for their different approaches to the thematic of software implemented fault injection.

2.4.1 Bond: An Agents-Based Fault Injector

Bond is a fault injection tool that is able to simulate abnormal behaviour within the Windows NT/2000 Operating Systems. These operating systems are important due to its use in the U.S. military, mainly in the Navy, since 1997.

This tool uses the call-based interface between the OS and the applications, Application Program Interface (API), and intercepts system calls in order to modify them without accessing the original code or writing device drivers. Building an intermediate interface, between the OS and the application layer, a specific agent is used to act as an extension of the OS for the application and as a wrapper separating the application from the OS [Baldini00].

Interposition agents record and, if necessary, alter the communication occurring between the user code (target application) and the OS code and these are not aware of the different environment. Through the use of these agents, it is possible to inject faults and to monitor the subsequent behaviour with a low overhead and with the added benefit of not modifying the OS or the user application.

The key objectives of the BOND tool are flexibility and stability. The latter is achieved by a careful implementation and low complexity. The former characteristic is based on a very adaptive fault model [Baldini00].

Architecture

The BOND tool uses two interposition agents, the logger agent and the fault injection agent. The former is responsible for monitoring the target application in order to trigger the injection phase and log the subsequent result. The latter is in charge of injecting the defined fault into the target application [Baldini00].

This approach allows for a greater simplicity in design and modularization, lowering the overall complexity of the injector and adding a degree of flexibility. On a multiprocessor platform these two agents could run in parallel and thus reducing the execution time overhead of the target application.

Logger Agent

The logger agent is in charge of synchronizing the fault injector agent with the target application and monitors the outcome of the fault injection. There three classes of events that this agent is able to detect:

- **Debug events**. For each debug event notified by the OS kernel, the agent records the target application context at that moment. This information includes the execution time, process and threads ID's and the processor's registers dump.
- **API calls**. Every time there is a communication between the target application and the OS (through an API call), the agent records the name of the call (function's name), its parameters and type, and an extract of the context that includes the execution time, process and threads ID's, return address and the return value of the function.
- **Memory accesses**. For each access to a certain region of memory, the agent records the virtual address of the bytes addressed, the access type, which can be read or write, the address of the instruction requesting the data and an extract of the context that includes the execution time, process and threads ID's. Every committed area of the process's memory can be monitored [Baldini00].

In order to make a decision of where to inject the fault, this agent performs a preliminary mapping of the target processor's memory. For each executable image, the injector identifies the different sections, with particular attention to the code and data sections. It also finds the position and the limit of the thread's stack as well as location of the main heap of the process and other allocated regions of committed memory, both local to the process and shared with other processes. [Benso03]

The two main tasks of the logger agent are the fault injection activation and the fault effects observation.

The fault injection activation depends on one assumption, the trade-off between precision and speed. This is due to the conceptual program structure that is built on the following premise, a simple on-line observation allows low overhead but also low precision, on the other hand a complex on-line observation typically implies a high monitored execution (could even be a step-by-step execution), with a clear decrease in performance.

This tool has two different types of fault injection:

- **Statistical injection,** a time-based non-deterministic approach where the injection is expressed as the elapsed execution time from the start of the application. The intrusiveness of this approach is minimal but at a cost: non-reproducibility of the experiment. This suitable for statistical studies that resort to a large number of injections, where the cardinality of experiments is important and these require a expedite experiments.
- **Deterministic injection,** which is a reproducible experimental injection method. It is based on the count of certain events to determine the exact moment to inject a fault.

This method allows for complex and precise experiments, however, it introduces a major overhead.

As one can see by the description above, statistical injection is based on timers and the deterministic injection on events. The usage of events to trigger the injection is based on a reliable synchronization with the logger agent, which provides real time information about the different events.

The logger agent provides the fault injector agent with two possibilities, i.e., operating modes. These are:

- **API mode**. This mode relies on the count of a specific API call. Faults can be injected both before the requested call and after it, in order to allow modification of the function's call parameters and return value.
- **Memory mode**. The event that triggers a fault injection in this operating mode is a specific memory access at a certain count. Since the amount of monitored memory greatly influences the overhead, the logger agent only observes the most critical areas.

After the fault injection, the logger agent monitors the behaviour of the target application and classifies the result accordingly to the following categories:

- **No effect**. This means that the injected fault had no effect and the result is the same to fault free experiment.
- **Last chance notification**. It falls into this category the experiments that lead the application to a forced termination and due to a non-recoverable exception.
- **Silent fault**. In this category the final state of the application is corrupt, i.e. it has different results from a fault free experiment, however it does not any effect on the program termination.
- **Error messages**. In this category there is a recoverable error message from the OS and a consequent error message.
- **Time out**. The application is forced to terminate due to a pre determined time out condition [Baldini00].

Fault Injection Agent

This agent is activated by the logger agent and it is responsible for injecting the determined fault. The conditions in which it does are three: location, type and duration.

BOND allows for injection into two particular locations:

- **Process's memory**. The fault injector agent can access the low part of the process's virtual memory, the lower 2 GB, which is reserved to the user and DLL's code and data. It should be pointed out that the reserved and free regions cannot be considered fault locations because they are not related to the physical memory. The fault injection agent is also able to inject fault into the process stack, allowing the possibility to inject parameters and return values to high level functions, such as the API calls.
- **Processor**. The thread context contains a copy of all the processor's registers. Modifying the context simulates faults appearing in the processor [Baldini00].

The BOND tool uses bit-flips as the only type of possible fault to be injected and the duration of these faults fall into two categories:

- **Transient**. These are triggered by environmental disturbances and are usually short in duration, however they are capable of corrupting the system.
- **Intermittent**. These cause the system to oscillate between periods of erroneous activity and normal activity [Baldini00].

2.4.2 MAFALDA-RT (Microkernel Assessment by Fault Injection AnaLysis and Design Aid of Real Time systems)

MAFALDA-RT is a SWIFI tool designed to characterise the behaviour of a microkernel in the presence of faults [Arlat02]. The objectives of this tool are based on one premise: SWIFI as a method to assess dependable systems with stringent real-time requirements raises the problem of temporal disturbances, caused by the overhead induced of SWIFI tools [Rodríguez02]. This is known as temporal intrusiveness. This effect is caused by the time related to the injections of faults and to the observation of the system behaviour.

MAFALDA-RT features two assessment capabilities, virtual elimination of the temporal intrusiveness as well as an enhancement of the observations made in the temporal domain [Rodríguez02].

The architecture of MAFALDA-RT consists of a set of target machines running a real-time system and a host machine. The latter is responsible for controlling and analysing the fault injections experiments.

On the target side, MAFALDA-RT comprehends 3 components (modules):

- **The Injector**. This component consists of a set of exception handlers that perform the fault injection. Only one fault type is used, bit-flip. The targets of fault injection are the code segments of the various components forming the micro-kernel. These can be, among others, the scheduler, the synchronization manager and the time manager [Rodríguez02].
- **The Observer**. This component consists of a set of data interceptors performing the observation of the target system. These interceptors capture information that enables the tool to properly identify the failure modes and the performance of the system.
- **The Device Controller**. This component controls system devices in order to erase the temporal intrusiveness introduced by the injection and observation of the system's behaviour. This is done by acting on the hardware clock.

On the host side the MAFALDA-RT one can find a database where all information pertaining the configuration of the tool as well as the results.

The MAFALDA-RT has clearly three fault models (with precise fault locations):

- Corruption of the input parameters of application system-calls (parameter fault injections).
- Corruption of the memory space of the microkernel (microkernel fault injection).
- Specific faults affecting semantic behaviour of the microkernel (notion of saboteur) [Benso03].

The first consists in applying a mask to the parameters of system calls in order to bit flip them and thus emulating the propagation of errors from the higher layer (application software) to the microkernel level.

The second consists of applying a mask in order to bit-flip the memory address space.

The third emulates software faults that can affect the behaviour of microkernel functions. This can be accomplished by changing the priority of a thread, by changing how semaphores or mutex restrict access to critical sections, among other actions.

As mentioned before, one of the problems that may arise from the use of SWIFI is the certain level of intrusiveness and its impact on the temporal behaviour of the target system, especially when dealing with real-time operating systems. The MAFALDA-RT has an approach that tends to virtually eliminate this overhead. It consists on stopping the evolution of time in the microprocessor while the tool executes. In this way the target real-time system has no perception of motion in time. In order to do this, the MAFALDA-RT functions based on the simple assumption, that real-time systems are interrupted driven and that even the notion of time is built upon interrupts. These interrupts can be divided in two categories, internal and external.

Internal interrupts are activated within the target system, for instance, clock interrupts control the release of periodic tasks in the system. External interrupts are those triggered by the external devices, for example, sensors [Benso03]. The used approach by the MAFALDA-RT consists in stopping all interrupts while it executes and resuming normal operation after.

The MAFALDA-RT is able to perform readings in order to be aware the fault was injected and its effect on the system. It characterises the fault manifestations in the following way:

- Not activated faults.
- Tolerated error by the system.
- Failure detected.
- Error detected.
- Error and failure detected [Rodríguez02].

2.4.3 FERARRI: a Fault and ERRor Automatic Real-time Injector

The FERRARI tool was devised in order to respond to a need for a flexible and powerful fault injection system. It is designed to emulate hardware faults so that the changes in the microprocessor are the same as if an internal hardware fault has occurred. This tool is capable of injecting transient errors and permanent faults [Kanawati92]. FERRARI's main features are the controllability over the time of injection throughout the execution of the application, its location, its type and duration, it is automatic, the ability to emulate hardware faults along with control flow errors, the ability to measure error detection latency and to locate an error that was either detected by an error detection technique or has led to a system failure [Kanawati92].

This tool comprehends four components (modules). These are the initializer/activator, the user information, fault injector, the collector/analyser and the management module.

Initialisation and Activation component

This module is responsible for preparing the target system for fault injection and to assess the result of a free fault injection. This extracted information comprehends the output of the program, execution time and the address space used by the execution of the program [Kanawati92].

User Information component

This component is responsible for gathering parameters supplied by the user. These include the experiment mode, FERRARI controlled/user controlled. The former calculates a pseudo-random time to inject the fault (time-based trigger) or a pseudo-random location to inject the fault (spatial-based trigger). In the latter, the user is asked for input about these parameters plus the duration and the bit-field mask. This tool is able to perform several types of faults. These are bit-flip, set a bit, reset a bit, set a byte and reset a byte.

There is a set of fault classes that can be chosen by the user, these are hardware, control flow and user defined.

The hardware class includes memory faults, external bus faults, faults in opcode decoding circuitry, faults in instruction pre-fetch circuitry and data registers [Schuette90].

Control flow class emulates faults in control bits and control flow. The former is about bit errors in instructions which results in executing a different instruction. The latter is about changing the value of the program counter, changing the target address of a branch instruction, or the execution of a trap instruction [Kanawati92].

In the user defined class, the user specifies a location in the source code as well as a condition for the fault injection. The condition is an evaluation of a Boolean expression in the application such as a loop index or a data variable. The fault is only injected when the execution of the program reaches the selected location and the condition is satisfied.

Fault Injection

As mentioned before, FERRARI supports the injection of permanent and transient faults. The mechanism is the same for both of them, the difference lies in the fault duration. In the case of a transient injection the duration is of one cycle. Permanent faults can last for several instruction cycles, or may span the entire execution interval of the application [Kanawati92].

Data Collection and Analysis

FERRARI logs every experiment that is performed. The data that is collected is the location of the fault, affected bit and affected register, if applicable. Besides this, it is analysed if the fault injected was dormant, i.e., did not induce any failure in the system, if an error was detected and if it led to a failure. The detection mechanism is logged, and if applicable, the error detection latency.

An analysis is performed by determining the count and the percentages related to coverage, latency, and type of error detection mechanism for each experiment [Kanawati92].

2.4.4 Exhaustif: A fault injection tool for distributed heterogeneous embedded systems

Exhaustif is designed to analyse systems in which is necessary to validate if reliability requirements are satisfied. This tool consists of two principal elements:

- EEM (Exhaustif Executive Manager)
- FIK (Fault Injection Kernel)

These components and their relation can be seen in figure 1.

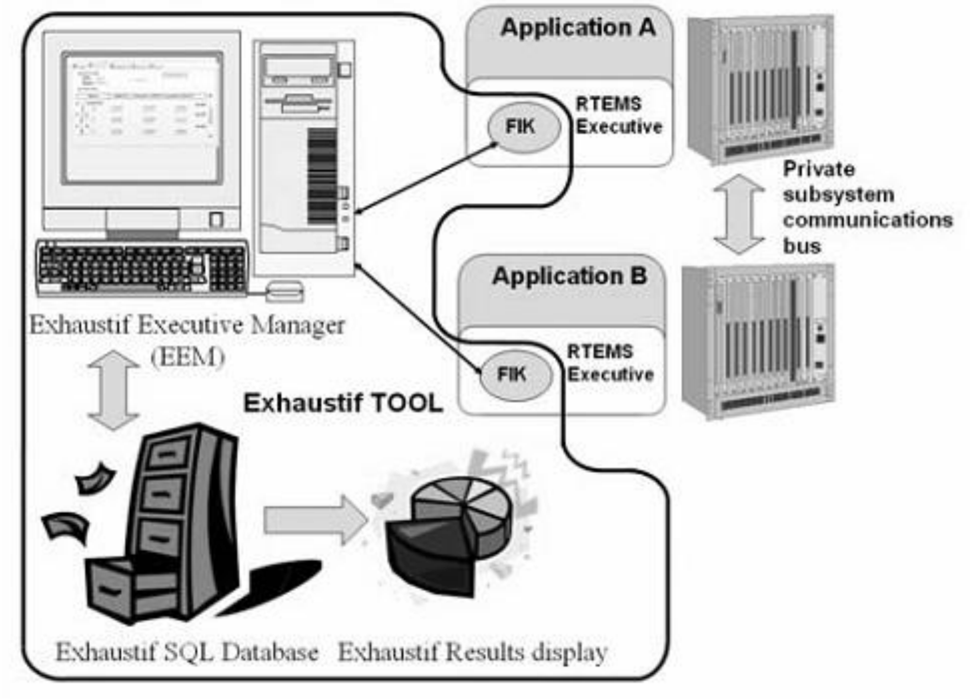


Figure 1 - Exhaustif architecture and components

EEM is responsible for control and management of all experiments. It provides the user with a graphical interface in order to specify the injection of faults and the visualisation of information of the experiments that were carried out.

FIK is a component that runs in the System Under Test (SUT) and it has the function of injecting faults [Dasilva07]. Exhaustif has the ability to support simultaneous SUTs. It allows for a more flexible usage, as it is possible to inject faults into several systems at the same time, which in turn form a bigger system, thus allowing assessing the results of insertion of faults in one subsystem in another one. This module is capable of injecting faults in the memory, CPU registers, Floating-point unit as well as intercepting function calls and alter its parameters and return values.

A standard SQL database is used to store all data produced within a fault injection.

2.5 Summary

In this chapter the objectives of fault injection were described, as well as the definition of dependability of a computer system along with its several facets, availability, reliability, safety, mean time to failure, coverage, maintainability, testability. Comparisons were drawn between two types of fault injections, hardware implemented and software implemented.

Four SWIFI tools were described, Bond, MAFALDA-RT, FERRARI and Exhaustif.

Bond can be used in Windows NT/2000 Operating System and resorts to two agents. One to log the fault injection outcome, the other one to perform the fault injections. The fault injections process can target a specific process's memory and the processor register.

The MAFALDA-RT is used to inject faults in a microkernel. It is composed of three components, the injector, the observer and the device controller. It can perform fault injections by corrupting input parameters of application system calls, by corrupting the memory space of the microkernel and by affecting the behaviour of the microkernel, such as altering the priority of a thread.

The FERRARI tool is composed of four modules, the activator, the user information, the collector and management. The fault injection that this tool performs are related to memory, external bus, opcode decoding circuitry, pre-fetch circuitry and data registers.

The Exhaustif tool is composed of two components, the exhaustif executive manager and fault injection kernel. This tool can inject faults in the memory, CPU registers, the floating-point unit and by altering the parameters and return values of function calls.

Chapter 3

Xception Tool

Technology does not drive change - it enables change.
Anonymous

The Xception tool enables a user with the ability for an advanced and effective validation, evaluation and testing of mission and business critical systems by emulating hardware faults through an ingenious use of the SWIFI technique. The description here presented concerns the general SWIFI plug-in for the PowerPC architecture.

Its architecture duplicates the generic Client/Server model. It comprises a front-end application, designated Environment Management System (EME), which runs in a host computer and is responsible for experiment management and control, and a lightweight injection core, which runs in the system being evaluated, designated Target System and is the responsible for the fault injection (figure 2).

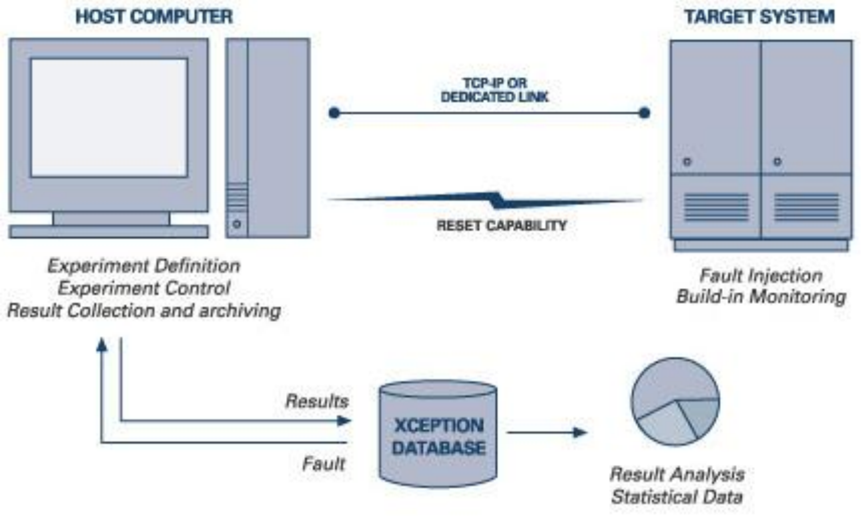


Figure 2 - Xception tool layout

Connection between the host and the target is done by means of a high level protocol, which stacks 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 [CSW03].

3.1 Host system: Experiment Manager Environment

The EME runs in the computer, designated as Host computer, and is responsible for the definition of faults, as well as the execution, control, outcome collection and analysis. A relational database is used to store the outcome of the performed fault injections. This database can be used to extract and cross important information in order to fully understand the gathered results.

Since the EME is developed in Java it can be executed in the main existing operating systems, such as Windows, Linux and Solaris.

3.2 Target System: Injection Run Controller

The Injection Run Controller (IRC) runs on the Target System and it holds all the necessary algorithms to emulate faults through software manipulation. This software module is installed in the Target System at the low exception handling level, performing fault injection in the most non-intrusive possible way.

Besides the injection core itself, there is a Communication Proxy (CP) that handles all communication with the Host computer. This communication comprehends all data concerning the fault injection process, i.e., fault parameters, outcome results, as well as target specific actions such as hard reset and system reboot.

3.3 Xception Features

It should be noticed that the Xception tool is considerably flexible and that the deployment for a specific target must require developing/porting, at the very least, the following modules:

- Target plug-in for the EME.
- The target specific IRC.
- The target Communication Proxy.

Taking this into account, from here on (unless otherwise noticed), everything that describes the Xception tool is related to the VxWorks-PPC603e plug-in.

3.3.1 Hardware fault emulation

In order to emulate hardware faults, the Xception tool uses two types of triggers: temporal and spatial. The temporal trigger causes the fault to be injected after a predetermined amount of time. A spatial trigger determines that a specific fault is injected only when a certain memory address is accessed or instruction is used. For instance, if the user wishes to inject a fault in the Integer Unit at a specific time/space, when it is triggered an integer type instruction is search and its content is changed (fault injection).

The fault locations of the Xception tool are all centralised in the microprocessor. These are:

- Integer Unit.
- Floating-point Unit.
- Memory Unit.
- Data Bus.
- Address Bus.
- General Purpose Registers.
- Floating Point Registers.

The type of fault that is performed by the Xception tool is bit-flip and all faults are transient.

The fault emulation comprehends several steps, depending on the location.

Integer Unit and Floating-Point Unit

Faults in these units are injected in three steps. First, it decodes the target register of the selected instruction (where the result will be deposited), then it executes the target instruction in trace mode, and afterwards corrupts the generated result in the target register.

Memory Unit

After the trigger is activated, the selected memory space is corrupted and processing is resumed.

Data Bus

Faults located in the data bus corrupt the data accessed, be it an instruction or a data item, after the trigger instant. Corruption is done by using the 32-bit mask, for instruction fetches and integer data accesses, and the 64-bit mask for double precision floating-point data accesses. If the fetched instruction is not a load or a store, then the instruction code is affected by the specified mask/type. If it is a load or a store instruction, then an additional fault parameter is used to select which access – code fetch or data –is to be corrupted.

Address Bus

Faults in the address bus corrupt the next address generated, be it for a code or data access, after the trigger instant. The fault mask used is always 32-bit.

If the fetched instruction is not a load or store, then the program counter is affected by the fault mask. From here we have three possible scenarios:

- If the new fetched instruction is not a jump, then the program counter is restored and incremented after that instruction.
- If the new fetched instruction is a call, then the return address is replaced by the incremented program counter initial value.

- If the new fetched instruction is a relative branch, then the program counter is replaced by its initial value plus the branch offset.

If the fetched instruction is a load or store, then an additional fault parameter is used to pick whether the address to be corrupted is the instruction address (same procedure as in the earlier case) or the data address (data address corrupted by the fault mask).

General Purpose Registers and Floating-Point Registers

Faults in these registers corrupt the target register at the trigger instant with a 32-bit mask, for the general purpose registers, and a 64-bit mask for the Floating-Point Registers.

3.3.2 EME Add-ons

The Xception tool is able to use two add-ons, the Easy Fault Definition (EFD) and Xtract modules. The former helps to define fault location and fault triggers by allowing the user to navigate through the source code and select and interactively mark memory ranges. This module provides the user with accurate and precise information, which in turn can be effectively be used to locate and fix many unsuspected bugs in the software under validation and verification.

The Xtract module is able to query the EME database in order to collect helpful data to non-SQL aware users. It already provides pre-defined queries whose output is placed in a formatted manner on a pre-determined format by the user, usually PDF. With this data the user can understand how the workload (application) responded to the injected fault, i.e., if it detected/undetected the fault and if it recovered or not.

3.4 Summary

This chapter describes the Xception tool. This tool is composed of two components, the experiment management environment and the injection run controller. The former is responsible for allowing the user to define faults and all parameters associated to it. The latter is in charge of performing the actual fault injections and colleting the results.

It can perform fault injections in the General Purpose registers, Floating-point registers, Integer Unit, Floating-point Unit, Data bus, Address bus and Memory.

Chapter 4

Porting Methodology

When I examine myself and my methods of thought, I come to the conclusion that the gift of fantasy has meant more to me than my talent for absorbing positive knowledge.
Albert Einstein

One of the main objectives of this project was to develop a methodology to port SWIFI plug-ins for the Xception tool. This methodology was defined to help the specific port of this project as well as any future port for SWIFI plug-ins. Before this methodology was fully defined, it was necessary to study the Critical Software's SDP and the Xception tool with the VxWorks-PPC603e plug-in.

4.1 Porting Methodology Overview

The objective of this methodology is to provide a set of ground rules and procedures that comprehends a porting methodology for SWIFI plug-ins for the Xception tool. Before delving into the methodology there is a set of concepts that should be clarified:

- What is porting?
- Need for porting.
- Porting and Software Development Process (SDP) interaction.

4.1.1 What is porting?

Many times it is quite difficult to determine the differences between porting and building. The latter can be defined as a previously defined process of creating an installable software package or applications, where as the former can be described as the software installation requiring undocumented changes to adapt it to a new environment. This means that this process can be as simple as building software for a new environment or a process which relies deeply on modification of the source code [Leheym95].

The effort that is necessary to port a software application can vary very much. It all depends on the similarities between the used Operating Systems (OS's) and hardware.

4.1.2 Need for porting

There are three main reasons for porting software. These are:

- **Different Operating System** – Depending on what features the OS offers, the software application may need to be modified. File name conventions, file system and system calls are among the most common differences.
- **Different hardware** – These differences can lead to major rewriting of source code, especially because SWIFI relies heavily on algorithms that use directly the hardware.
- **Local choices** – This includes installation pathnames and cooperation with other installed software. [Leheym95]

4.1.3 Porting and Software Development Process interaction

There is an already defined process at Critical Software (CSW) to develop software. It is clearly defined in the company's Quality Management System. The general SDP used at CSW consists of five phases, as is shown in figure 3.

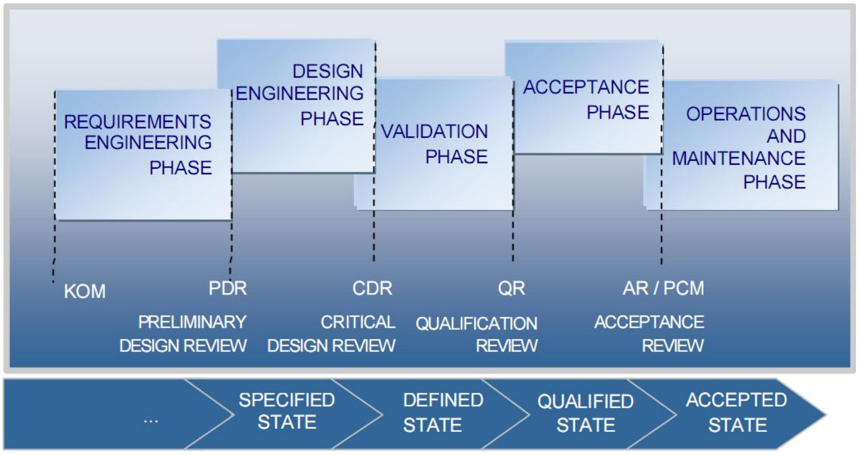


Figure 3- Critical Software's Software Development Process

The first phase is the Requirements Engineering Phase that starts with a Kick-Off Meeting where the project plan is presented, a risk analysis is performed and all team members are introduced and their roles within the project are defined. This phase consists on the elaboration of technical specification that contains precise and coherent definition of performances, cost, schedule and implementation plans for all levels of the software to be developed. The Software Requirements Specification (SRS), Test Case Specification (TCS) and the Software Architecture Specification and Database Design Specification (DBS) documents are produced in this phase. In this phase, it is not supposed to exist any kind of implementation, unless a prototype is needed to aid the specification process. The Requirements Engineering Phase ends in the preliminary design review (PDR) meeting. On this meeting all the produced documentation must be validated.

The second phase is the Design Engineering Phase during which the implementation is done, consistent with the documentation made in the previous phase. More specifically, on this phase it is produced the detailed design, the software code and the unit test of each element of the software product, in response to the requirements contained in the technical specification. This phase ends in the Critical Design Review (CDR) meeting which purpose is to validate the source code and testing that was performed.

Next is the Validation phase, where the test cases defined in the first phase are run and the code is evaluated as being or not in conformity with the specification. This phase ends in the qualification review (QR) meeting in which the project manager makes sure that an analysis of the system testing results and a usability review was made.

The next two phases focus on the client and the relation with it, being the Acceptance phase where the client accepts the developed product and the Operations and Maintenance phase used for the cases where a maintenance deal is made. At the end of the Acceptance Phase an Acceptance Review (AR) and Project Closedown Meeting is performed. The purpose of these two steps is to review what happened in the project (if all objectives were achieved and the customer needs were fully met). On this meeting it should be planned the warranty of the produced software and the maintenance plan.

As it can be seen by figure 3, this SDP resembles the very well known Waterfall SDP [Sommerville07] with some overlaying between phases in order to allow for a smoother flow.

The porting methodology reflects these software development life cycle phases. It should be noticed that this porting methodology had to be clearly defined under the scope of Critical Software's SDP, as such this limited the approach i.e., other development processes were not considered. A set of inputs and outputs were defined for each phase that could jumpstart the beginning of the following phases.

These inputs and the outputs differ from the general SDP because of the very nature of the port, i.e., it already has a set of requirements defined, as well as design and, obviously, implemented base source code.

The inputs are artefacts which provide a foundation, regarding necessary information, to develop the output artefacts.

The following sections will focus upon each one of these phases and artefacts and explain them in detail.

In table 1 it can be observed the several phases of the porting process, its inputs and outputs. The acronyms are not described because it would make the table unreadable. Nevertheless it should be noticed that the term original and destination describes, respectively, the actual existing implementation and the one that will result from the port and the "tpl" subscript stands for template, i.e., a common artefact in all ports where as "specific" subscript identifies specific artefacts of the port at hand.

	Requirements Elicitation Phase	**Design Phase**	**Implementation Phase**	**Acceptance phase**	**Operations and Maintenance Phase**
Inputs	SWIFI Methodology (Fault Model and General Require-	DDS_{tpl}	SRC_{tpl}	$TCS_{specific}$	Maintenace Plan

	Requirements Elicitation Phase	Design Phase	Implementation Phase	Acceptance phase	Operations and Maintenance Phase
	ments)				
	Original Operating System Specification	UM_{tpl}		TER_{tpl}	
	Original Hardware Programming Manual	IM_{tpl}			
	Original Hardware User Manual				
	Destination Operating System Specification				
	Destination Hardware Programming Manual				
	Destination Hardware User Manual				
	SRS_{tpl}				
	SAS_{tpl}				
	DBS_{tpl}				
	TCS_{tpl}				
Outputs	$SRS_{specific}$	$DDS_{specific}$	$SRC_{specific}$	$TER_{specific}$	$MP_{specific}$
	$SAS_{specific}$	$UM_{specific}$	$DDS_{specific-updated}$	$UM_{specific-updated}$	
	$TCS_{specific}$	$IM_{specific}$		$IM_{specific-updated}$	
	$DBS_{specific}$				

Table 1- Porting Methodology Artefacts

The information on table 1 can be seen as a process similar to the SDP on figure 4 and more precise description of each phase can be seen in figures 5, 6 and 7.

Notice that Requirements, Design and Implementation phases can overlap, depending on the specific port. It should be pointed out as well that the output of the Operations and Maintenance depends largely on the type of maintenance. For instance, if it is a type of maintenance that requires a new functionality it may mean a whole new porting process (see section 4.9 for more details). Due to the nature of this SDP, the Design phase handles the design analysis and leaves the implementation for the next phase. The general Validation and Acceptance phase are put together in the Acceptance phase.

One could argue that the Operations and Maintenance phase is not suitable for a porting process, however it was considered important to consider it in order to respect the original SDP and to mention the subtleties of the maintenance process in a SWIFI plug-in.

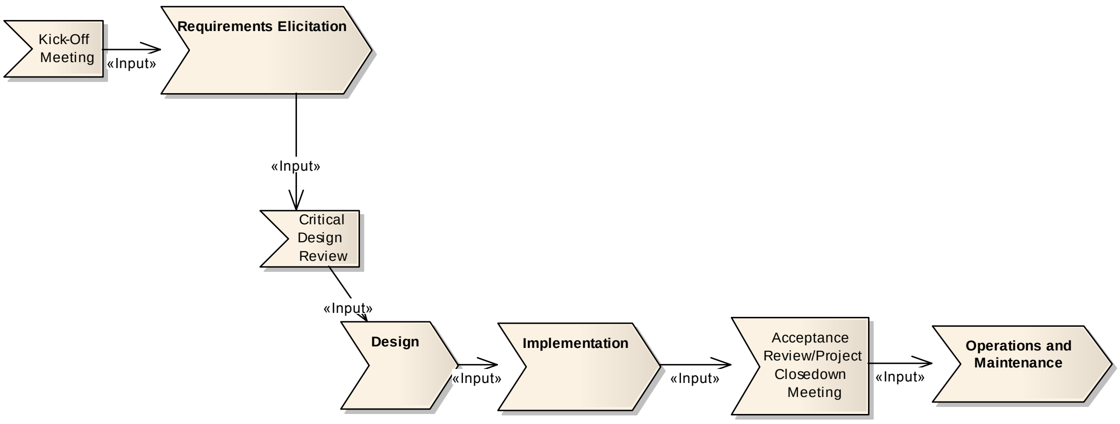


Figure 4 - Porting Methodology Sequence

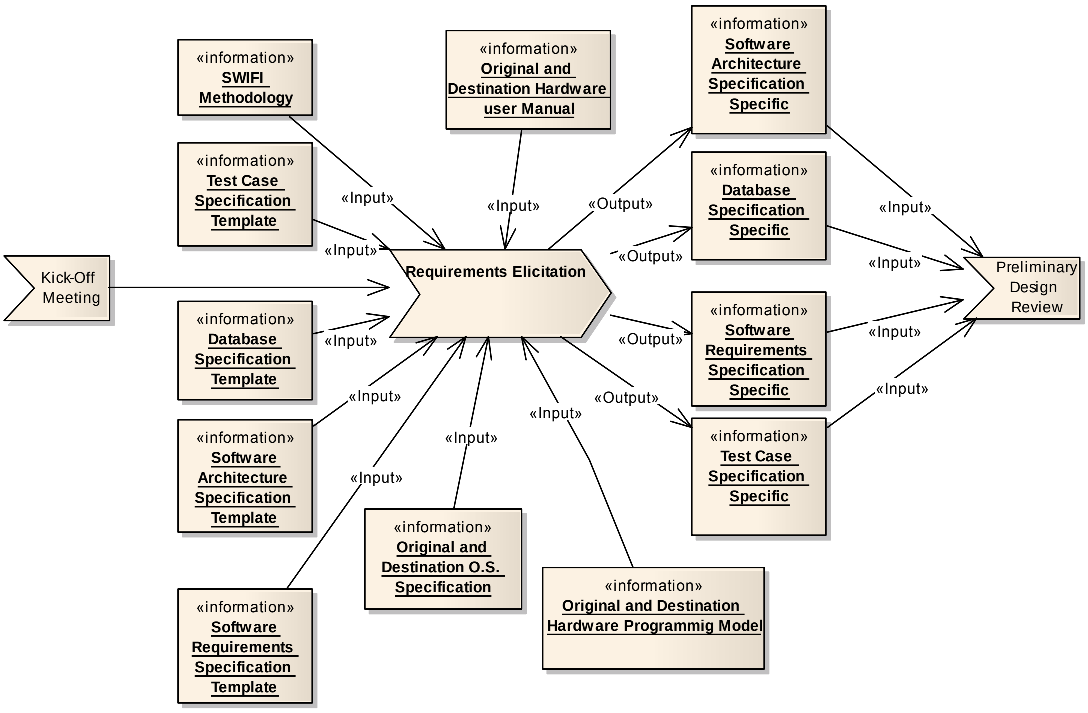

«information» SWIFI Methodology
«information» Test Case Specification Template
«information» Database Specification Template
«information» Software Architecture Specification Template
«information» Software Requirements Specification Template
«information» Original and Destination Hardware user Manual
«information» Original and Destination O.S. Specification
«information» Original and Destination Hardware Programmig Model
Kick-Off Meeting
Requirements Elicitation
«Input»
«Output»
«information» Software Architecture Specification Specific
«information» Database Specification Specific
«information» Software Requirements Specification Specific
«information» Test Case Specification Specific
Preliminary Design Review

Figure 5 - Requirements Elicitation Phase

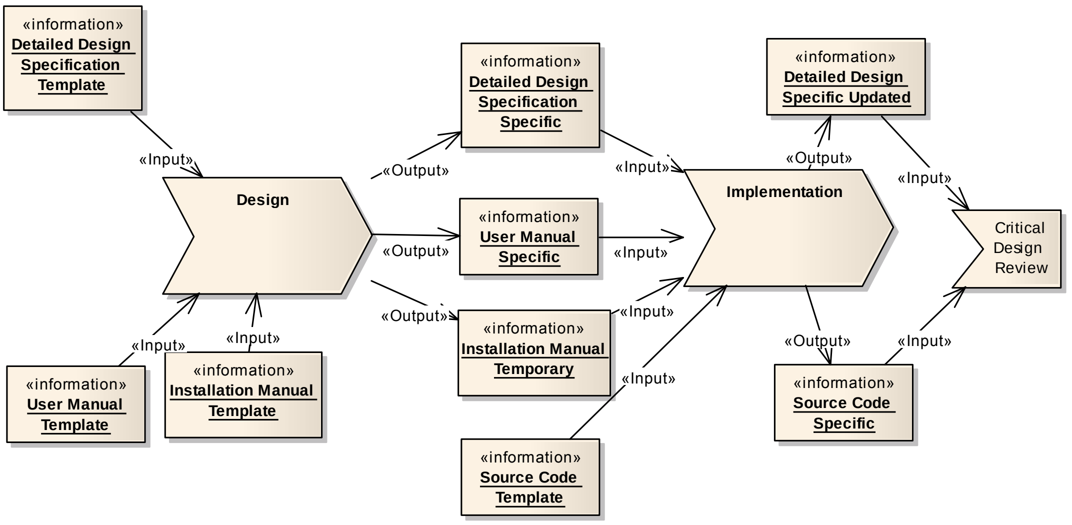

«information» Detailed Design Specification Template
«Input»
Design
«Output»
«information» Detailed Design Specification Specific
«information» User Manual Specific
«information» Installation Manual Temporary
«information» User Manual Template
«information» Installation Manual Template
«information» Source Code Template
Implementation
«information» Detailed Design Specific Updated
«information» Source Code Specific
Critical Design Review

Figure 6 - Design and Implementation Phases

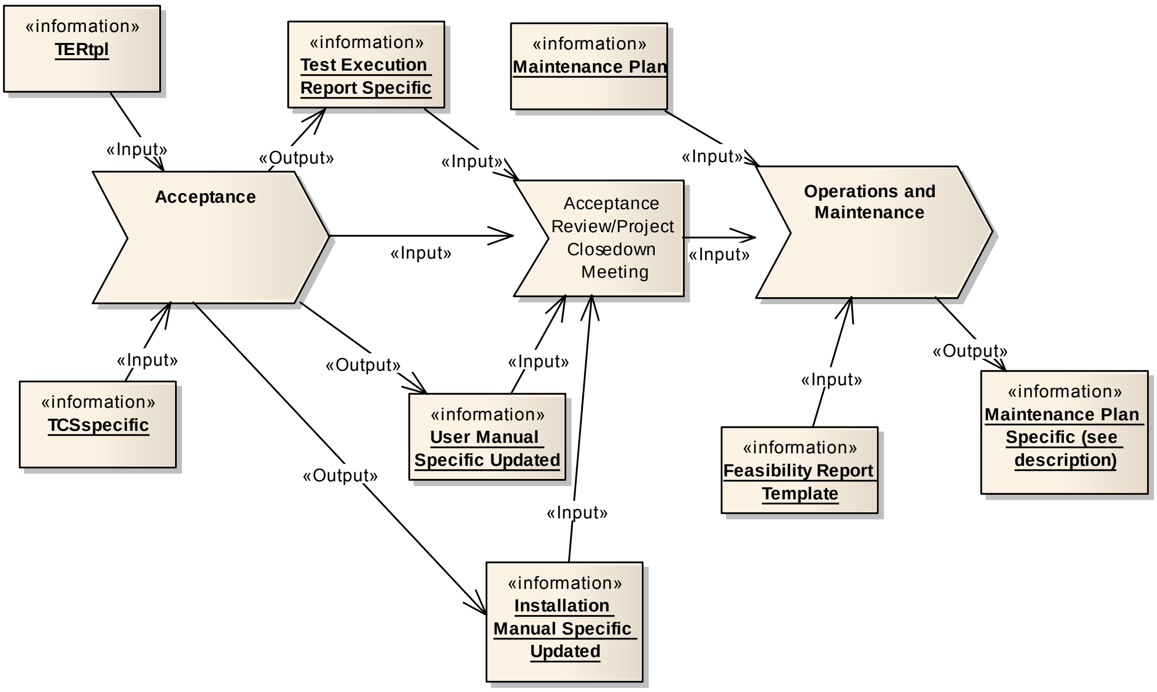

«information» TERtpl
«information» Test Execution Report Specific
«information» Maintenance Plan
«Input»
«Output»
Acceptance
Acceptance Review/Project Closedown Meeting
Operations and Maintenance
«information» TCSspecific
«information» User Manual Specific Updated
«information» Installation Manual Specific Updated
«information» Feasibility Report Template
«information» Maintenance Plan Specific (see description)

Figure 7 - Acceptance and Operations and Maintenance Phases

In the following sub-sections, all of these phases will be described as well its input and output artefacts.

4.2 Requirements Elicitation/Specification Phase and Input Artefacts

This phase is responsible for the process of understanding and defining what is needed in order to port the plug-in. This stage is particularly critical since any error that occurs at this point in the process will affect adversely all following phases and may lead to an unsuccessful port.

The team members involved in project should take into consideration the following input artefacts:

- SWIFI methodology;
- Original and Destination Operating System Specification;
- Original and Destination User Manual and Programming Manual;
- SRS, SAS, DBS and TCS template documents.

4.2.1 SWIFI methodology

The SWIFI methodology describes the general concepts behind this type of validation tool. It is the cornerstone of the plug-in. It answers all questions pertaining where to inject faults, what triggers these injections and what data to collect and analyse. These two last points are extremely important in order to provide a useful tool for the end user.

It also describes de fundamental requirements a plug-in should comply with in order to meet the necessary criteria for obtaining reliable and credible measurements.

The SWIFI methodology comprehends a base theoretical model and a set of general requirements that should apply to all plug-ins.

Theoretical Model

The FARM model [Arlat90] was developed in the early 90's at the Laboratory of Systems Analysis of the French National Centre for Scientific Research and is widely recognized as the *de facto* theoretical model to describe any SWIFI methodology. It has the following set of attributes:

- The **F** corresponds to the **set of faults** which can be described as the input domain.
- The **A** is the **set of activation** that specifies the domain used to functionally exercise the system;
- The **R** is the **set of readouts** which can be described as the output domain;
- The **M** is the **set of measures** of the system's dependability which are obtained by an analysis of the results of fault injection. [Arlat90]

This model can be improved by including the **set of workloads W** [Benso99]. The workload can be a real world application, a benchmark or a synthetic application.

SWIFI General Requirements

Although most of the requirements are constrained by the OS/Hardware Architecture combination, there is a set that should always be considered and a plug-in should always comply with.

These requirements are:

- Representative fault model and fault selection (Faults):
    - In order to draw conclusions of a system's dependability, the faults injected must be representative of the real faults.
- Low intrusiveness (Activation):
    - Intrusion introduced by the fault injection environment may cause delays and compromise the real functioning of the system, providing non-real results.
- Observability (Readouts):
    - It must be possible to collect information in order to evaluate the system and calculate measures.
- Inspection (Measurements):
    - The measures of the fault injection process must be transparent and auditable. This means that the results can be achieved by repetition and there is an unequivocal storage of results.[Benso03]

4.2.2 Original and Destination Hardware Programming Model – Programming Model

It is essential to analyse the Programming Manuals in order to understand what the capabilities the hardware provides. Special attention should be given to:

- Instruction set;
- Register set and conventions;
- Memory paradigm;
- Exception paradigm;
- Memory management paradigm;
- Time paradigm.

Instruction Set

The instruction set usually describes which instructions exist, the existing format, forms of encoding, addressing modes, algorithmic description of operations, the effect of flags and operands.

Register Set and Conventions

This should comprehend what registers exist, memory conventions, bit and byte ordering and how data is stored.

Memory Paradigm

It should be studied a set of memory related subjects. These are related to how the size of address space and its subdivisions are made, how to configure pages and blocks of memory, as well as the several types of memory protection. Memory management should also be taken into account. This includes descriptions of how the memory can be partitioned, configured, protected, as well as how memory translation is performed and special memory control instructions.

Exception Paradigm

The set of exceptions and its properties should be studied in order to understand how they work and how they can be used in order to access important microprocessor's functions.

Time Paradigm

How the microprocessor handles time should be studied in order to understand how it determines the time of the day and what mechanisms exist for time-related exceptions.

4.2.3 Original and Destination Hardware Programming Model – User Manual

By studying the hardware's User Manual the responsible for the port should become acquainted with specifics about the microprocessor and its capabilities. Typically, much of the information found in these manual can also be found in the Programming Manual. Nevertheless, these manuals include precious information about the specifics of microprocessor, and in many cases, differ from what is defined in the more generic Programming Manual. Special attention should be given to:

- Specific architecture implementation.
- Instruction set.

Specific Architecture Implementation

Each microprocessor has its own particular characteristic that does not share, even with microprocessors of the same family. It is very important to check its features, which may include, among other properties, the following:

- Execution units.
- System performance.
- Power management.
- Signals.
- Instruction timing.
- Cache implementation.

Instruction Set

Although the Programming Manual should describe every possible instruction the hardware is capable of, the User Manual should describe if it has any specific instructions and/or instructions that are not implemented.

4.2.4 Template Documents

There is a set of Template Documents that should be used in order make a parallelism to the SDP. This will allow for a better and more precise development further along in the porting process.

These documents have precise guidelines to expedite the process of writing a concise, yet precise document regarding everything about the development of the plug-in.

These documents are:

- Software Requirements Specification Template (SRS_{tpl});
- Software Architecture Specification Template (SAS_{tpl})
- Database Design Specification Template (DBS_{tpl})
- Test Case Specification Template (TCS_{tpl})

4.2.5 Software Requirements Specification Template

The SRS_{tpl} should contain the necessary requirements and these should be analysed and detailed in a way that can efficiently be translated to the project, it also means that the level of detail should be adapted to the particularities (constraints and goals) of the plug-in. [Sommerville07]

Nevertheless, there is a set of requirement guidelines that this document should include in order to bind the writer to adequate requirements logic for an Xception plug-in. It should be taken into account that these guidelines can be ignored if they do not apply to a certain plug-in. However this should not be done lightly since they can be quite helpful.

These guidelines are:

- Description of how the communication is handled between the plug-in and the Xception Frontend Experiment Management Environment (EME);
    - Set of messages that the Plug-in must be able to receive and interpret correctly;
    - Set of messages the Plug-in must be able to send accordingly to requests of the EME;
- Description of how the communication is handled between the host side of the plug-in and the target side;
    - Set of messages that both sides must be able to send /receive and interpret correctly for a correct fault injection process;
    - Format of exchanged messages;
    - The format and interpretation of these messages can be defined within an ontology so that each side knows in which context the messages are being exchanged. For instance, the host system could send a reset request in various scenarios. One of these scenarios could be a reset request after a fault injection. If the target has not yet performed a fault injection it would be considered out of context and lead to an error. This contextualization within an ontology allows for a better debugging and assurance that the defined process is performed correctly.
- Fault Location

    - Definition of locations to inject faults and if applicable determine special cases. This should take into consideration the SWIFI general requirement concerning representativeness.
- Fault Triggers
    - According to the SWIFI methodology the fault triggers shall be defined how they should work. Usually the triggers are temporal and spatial. This should take into consideration the SWIFI general requirement concerning representativeness.
- Fault types
    - Definition of type of fault. This should take into consideration the SWIFI general requirement concerning representativeness.
- Data collecting
    - Before and after a fault is injected, information should be gathered about the state of the microprocessor and the output of the application that is running (workload). This should take into consideration the SWIFI general requirement concerning observability.
- Resources
    - The SWIFI technique has good results due to many factors including the low level of resources usage during the fault injection process. This should be taken into account. This should take into consideration the SWIFI general requirement concerning low intrusiveness.
- Fault injection organisation
    - The Xception tool has by default a tree approach to organise the fault injection process. If that is to be used, include the necessary requirements that mention “Campaigns”, “Experiments”, “Injection Runs” and “Faults”.
- Report generation
    - It should be possible to generate reports so the user can assess the results of the fault injections. This should take into consideration the SWIFI general requirement concerning inspection.

4.2.6 Software Architecture Specification Template

The $\mathrm{SAS}_{\mathrm{tpl}}$ document should describe the context of the system, give a system overview, identify restrictions/principles that will drive all architectural decisions, deployment, describe a system decomposition and describe a structure through a package diagram.[Sommerville07] Following the guidelines here describe should help create an architecture rapidly.

These guidelines are:

- Definition of a high-level architecture describing the components and how they are connected. These connections are very important to understand the relations between the host and target system as well as with other plug-in add-ons.
- Definition of packages and its relations within the host and target system;
- Definition of packages and its relations within the host system;

- Design of a state machine for the complete fault injection process for better debugging.
- Design of sequence diagrams to explain the exchange of messages within the host system and with the target system.

4.2.7 Database Design Specification Template

The DBS_{tpl} document must describe precisely the necessary databases, the relation between information (tables), its description and importance to the system.
The guidelines for this template concern basic information that should be kept.

These are:

- Target System;
    - Identification of target system;
- Fault;
    - If fault was injected, its location and trigger;
- Workload outcome;
    - Workload output and time;
- Fault outcome;
    - Logs of the microprocessor state before and after fault injection;

4.2.8 Test Case Specification Template

The TCS_{tpl} document should describe the process which will verify the software to determine compliance with the defined requirements. This way it will be possible to increase the plug-in's reliability and to show beyond any doubt it performs as it is supposed.

The guidelines for this template are:

- Test specific critical exchange of messages.
- Test fault injection algorithms.
- Test proper results generations.

4.3 Requirements Elicitation and Specification Phase Output Artefacts

The output should comprehend the following artefacts:

- Software Requirements Specification Specific to the new plug-in ($SRS_{specific}$)
- Software Architecture Specification Specific to the new plug-in ($SAS_{specific}$)
- Database Design Specification Specific to the new plug-in ($DBS_{specific}$)
- Test Case Specification Specific to the new plug-in ($TCS_{specific}$)

These artefacts are specific to the plug-in under development and close de Requirements Elicitation and Specification phase. They will be used for development in the following phases.

One should notice that like a usual SDP they can be updated and changed further down in the development chain.

4.4 Design Phase and Input Artefacts

This phase differs from the usual SDP. The general idea is to analyse the design of the existing plug-in and how it complies with the specific documents developed in the previous phases. The beginning of this phase may overlap the end of the previous one in order to achieve a better accordance with the mentioned documents. This phase allows for a deep understanding of the existing plug-in as well as the full potential of the port.

The team members involved in project should take into consideration the following input artefacts:

- Detailed Design Specification Template (DDS_{tpl});
- User Manual Template (UM_{tpl});
- Installation Manual Template (IM_{tpl})

4.4.1 Detailed Design Specification Template

The DDS_{tpl} document should explicitly include all packages that belong to the software along with precise and detailed information about all features. As such, this document should become a reference for the development team to port the plug-in successfully.

Taking into account this is a design document for a port, the approach may vary from a more general development. First and foremost, the design of the port will be strongly influenced by the existing one, therefore special care should be taken in the analysis of the original plug-in.

As such it is advisable to take into consideration the following guidelines:

- Team members that are responsible for design should be or get acquainted with a reliable source code documentation tool;
- Explicitly divide the design analysis of the host system and target system;
- If applicable, divide the design analysis of the target system in semi or completely portable code, for instance C code, and hardware specific code, i.e., assembly;
- Trace implemented algorithms/functions to requirements.

4.4.2 User Manual Template

The User Manual Template aims to describe the steps the user must follow to guarantee a correct use of the plug-in.

The following guidelines should be respected:

- Include an explanation of all functionalities of the plug-in, as well as a complete example of how to make a fault injection and consult its results.
- Include an example of how the client can include its own applications. Alternatively make reference to documentation that could provide similar information.

4.4.3 Installation Manual Template

The Installation Manual Template aims to describe the steps the user must follow to guarantee a correct installation of the plug-in.

The following guidelines should be respected:

- Explain in a very detailed way how to install the whole system, giving special attention to the development environment and if applicable to the hardware.

4.5 Design Phase Output Artefacts

The output should comprehend the following artefacts:

- Detailed Design Specification Specific ($\mathrm{DDS}_{\mathrm{specific}}$)
- User Manual Specific ($\mathrm{UM}_{\mathrm{specific}}$)
- Installation Manual Specific ($\mathrm{IM}_{\mathrm{specific}}$)

These artefacts are specific to the plug-in under development and close the Design phase. They will be used for development in the following phases. One should notice that like a usual SDP they can be updated and changed further down the development chain.

4.6 Implementation and Input Artefacts

In this phase, the original plug-in source code is modified in order to comply with the SWIFI methodology, more specifically to what was defined in the $\mathrm{SRS}_{\mathrm{specific}}$ and respecting the defined architecture and design which, subsequently, were based on the differences/comparison made between the OS's and the target systems. The presented guidelines are for the only input artefact, Source Code Template ($\mathrm{SRC}_{\mathrm{tpl}}$).

4.6.1 Source Code Template

The guidelines for this artefact are:

- Work your way up from the target system to the host system. This way the hardest problems are addressed – Hardware and target OS differences.
  - Adapt the most possible assembly language. Incompatible hardware accesses that are made should replaced accordingly to the new microprocessor architecture. The most common incompatibility is, besides instructions, specific microprocessor register accesses;
  - It is possible that the original plug-in uses values specific to the hardware, for instance, microprocessor's clock speed. If those values are needed they should be replaced by the proper ones;
  - System calls that are made should be assessed. Although it is advisable to use the least possible or none at all, sometimes programmers have no choice but to use them. Therefore, all system calls should be replaced with appropriate ones for the new OS;

    - The process in which information is kept in dynamic memory should be assessed in order to understand if the new target hardware has the amount necessary to support it;
    - While keeping the information in dynamic memory and at the same time reusing already defined variables can lead to illegal accesses;
    - The way the communication is performed should be assessed and changed accordingly. This may include communication protocol and message format.
- The source code applicable to the host system should be analysed from the communication end with the target system towards the communication end with the EME and the database.
    - The communication that is performed should be assessed. As mentioned in the previous sub-point, the communication protocol and message format should be checked;
    - Communication with the EME should be assessed in order to make sure the fault injection process is performed correctly and there are no out of order message exchanges;
    - Communication with the database should be assessed in order to make sure that loads and saves are performed correctly and comply with the defined fault model, fault definition and outcome results.

4.7 Implementation Phase Output Artefacts

The output should comprehend the following artefacts:

- Source Code Specific ($SRC_{specific}$);
- Detailed Design Specification Specific Updated ($DDS_{specific-updated}$);

These artefacts are specific to the plug-in under development and close de Implementation phase. They will be used for development in the following phases. One should notice that like a usual SDP they can be updated and changed further down the development chain.

The $DDS_{specific-updated}$ should be produced in order to update the $DDS_{specific}$ so that it reflects the implemented code.

4.8 Acceptance Phase and Input Artefacts

In this phase it should be demonstrated that the porting was successful and complies with the defined requirements. To perform this, tests should be executed and if any errors are found they must be corrected. The tests to be run are already defined in the $TCS_{specific}$.

The input artefacts for this phase are:

- Test Case Specification Specific ($TCS_{specific}$);
- Test Execution Report Template ($TER_{template}$);

Since the $TCS_{specific}$ was already defined and described in the Requirements Elicitation, only the $TER_{template}$ is described in this section.

4.8.1 Test Execution Report Template

This document, Test Execution Report Template, should contain the tests that were run (described in the $\mathrm{TCS}_{\mathrm{specifc}}$), its results, identified faults, and an overview of the test results in order to understand the readiness of the plug-in.
The guideline for this document is:

- Due to the nature of the fault injection process, the criticality level of errors must be pondered carefully. Anything that prevents the fault injection process from functioning correctly should be considered a serious fault that compromises the use of the system or relevant part of this system. This process includes communication, fault definition, fault triggers and location, results logging and saving information into database.

4.9 Acceptance Phase Output Artefacts

The output should comprehend the following artefacts:

- Test Execution Report Specific ($\mathrm{TER}_{\mathrm{specific}}$).
- User Manual Specific Updated ($\mathrm{UM}_{\mathrm{specific\text{-}updated}}$).
- Installation Manual Specific Updated ($\mathrm{IM}_{\mathrm{specific\text{-}updated}}$).

These artefacts are specific to the plug-in under development and close de Acceptance phase. One should notice that like a usual SDP they can be updated and changed further down the development chain.

The $\mathrm{UM}_{\mathrm{specific\text{-}updated}}$ and the $\mathrm{IM}_{\mathrm{specific\text{-}updated}}$ should be produced in order to update, respectively, the $\mathrm{UM}_{\mathrm{specific}}$ and $\mathrm{IM}_{\mathrm{specific}}$ so that it reflects the necessary adaptations that rose from the tests execution.

4.10 Operation and Maintenance and Input Artefacts

In this phase the plug-in should, if necessary, be modified to correct faults, improve features or add new ones. This should be done while preserving the plug-in's integrity.
The input artefacts for this phase are:

- Maintenance Plan Document (MP);
- Feasibility Report Template Document ($\mathrm{FR}_{\mathrm{tpl}}$).

4.10.1 Maintenance Plan Document and Feasibility Report Template Document

These two documents are in the same section because they are very dependent of each other
Before describing the guidelines one should be aware of the types of maintenance there are under the CSW's QMS:

- Preventive Maintenance (PM). This type should indentify and detect latent faults as well as identify safety concerns;
- Emergency Maintenance (EM). This type of maintenance is to be applied with unscheduled errors.
- Corrective Maintenance (CM). This type of maintenance should identify and remove detected defects as well as it correction of reported errors.
- Perfective Maintenance (PeM). This kind of maintenance should be performed to improve features and add functionalities.
- Adaptive Maintenance (AM). This kind of maintenance should be performed to adapt or upgrade the plug-in to a new environment.

Taking into account the previous description the following guidelines should be considered to develop the MP and the responsibilities towards the client as well as the concerns to have in the development of a FR:

- The PM, EM and CM types of maintenance should be dealt as in any normal project. They are focused on bug fixing and these bugs should be traced until the phase it occurred and the respective outputs should be changed accordingly. For example, if a requirement must be changed, everything that is affected by it must be changed as well.
- The PeM and AM types of maintenance are a special case in the porting process and should always be accompanied with a careful feasibility report.
    - In the PeM case, improving new features or even adding new functionalities could seriously affect assumptions made in the Requirements Elicitation phase. Although at first hand a new functionality might not seem much, if analysed in a greater detail it could tear down suppositions that were rock solid and as such, be unfeasible. This means that a new feature could demand a change in the SWIFI Methodology and change completely the philosophy of the plug-in, i.e. the Fault Model (see 4.2.1), and therefore all subsequent outputs. It could even infringe the basic requirements of SWIFI (see 4.2.1).
    - In the AM case, to adapt to a new environment, hardware, OS or both means a whole new porting process. A new study would be made and depending on the outcome it could mean a full-fledged port or simply a change in some features.

4.11 Operations and Maintenance Output Artefacts

The output should comprehend the following artefact:

- Maintenance Plan Specific.

This artefact should describe in what way the maintenance of the plug-in should be performed, accordingly to the feasibility report and the type of maintenance.

4.12 Summary

This chapter describes the porting methodology that was defined to provide a basis for the actual port. This methodology was based on the Critical Software's software development process with some variations. It is composed of the following phases: Requirements Elicitation, Design, Implementation, Acceptance and Operations and Maintenance.

Each one of these phases has a set of inputs and outputs that are a base for the following phase. There are also guidelines for each phase that aid the port.

In the Requirements Elicitation the original and target hardware's and operating system's manuals should be studied, a SWIFI methodology should be defined and the requirements, architecture, tests and database documentation should be developed.

In the Design phase, it should be produced a detailed design specification as well as the user manual and a temporary installation manual.

In the Implementation phase the source code for the new plug-in should be produced and an update to the detailed design should be made.

In the Acceptance phase tests should be performed, and a test execution report reflecting these tests should be produced.

In the Operations and Maintenance phase a maintenance plan and a feasibility report should be produced. The latter should describe if a certain type of maintenance is possible and if it respects the plug-in SWIFI methodology.

Chapter 5

Port of the Xception SWIFI Plug-in VxWorks-PPC603e

Low-level programming is good for the programmer's soul.
John Carmack

In this chapter, the actual port of the VxWorks-PPP603 plug-in to the PowerPC 750 Microprocessor, named VxWorks-PPC750, is presented. It should be said that the original plug-in (VxWorks-PPC603) lacked much of the standard documentation, i.e., requirements, architecture, database and design. This led to a time consuming writing of the several inputs, especially in the requirements elicitation phase. However, this will lead to an easier usage of the porting methodology in future ports, due to its reuse.

In order to avoid overwhelming the reader, it will not be presented every single detail of all artefacts. Only the necessary to understand what was effectively made and how, and to give a general idea of the approach.

5.1 Requirements Elicitation Phase

5.1.1 SWIFI Methodology

The very first point on this phase was to define the SWIFI methodology used on the future plug-in. This was of paramount importance in order to create a context of usage for the plug-in. One could argue that if the plug-in already existed then its context of usage was already defined. While this is true, by contextualising its usage in terms of applicability and how it fits the *de facto* theoretical model helps to determine the necessary requirements as well as define in a very precise and logical way the implementation, particularly the very complex injection algorithms.

A very important advantage of this definition is to prevent developers, who have to deal with this sort of plug-in, of going through the ordeal of understanding the very complex code as well as the concepts underlying SWIFI which is of paramount importance for a successful port/implementation.

Some of the descriptions here presented can be seen repeated elsewhere in the document. Nevertheless, although partially redundant, for the sake of integrity, all items of this methodology shall be described here with more information.

The figure 8 depicts the general overview of the SWIFI methodology. It is divided in three main pillars, Domain Analysis, Fault Model and Xception SWIFI. The Domain Analysis comprehends the object of test and the reasons behind these tests, the advantages of SWIFI over its main alternative and its objectives. The Fault Model comprises the theoretical and general requirements that should be taken into account when developing a SWIFI plug-in for the Xception tool as well as how this kind of plug-ins fit to those two items. The Xception SWIFI describes how the fault model fits the two previous items and how it works in a general way.

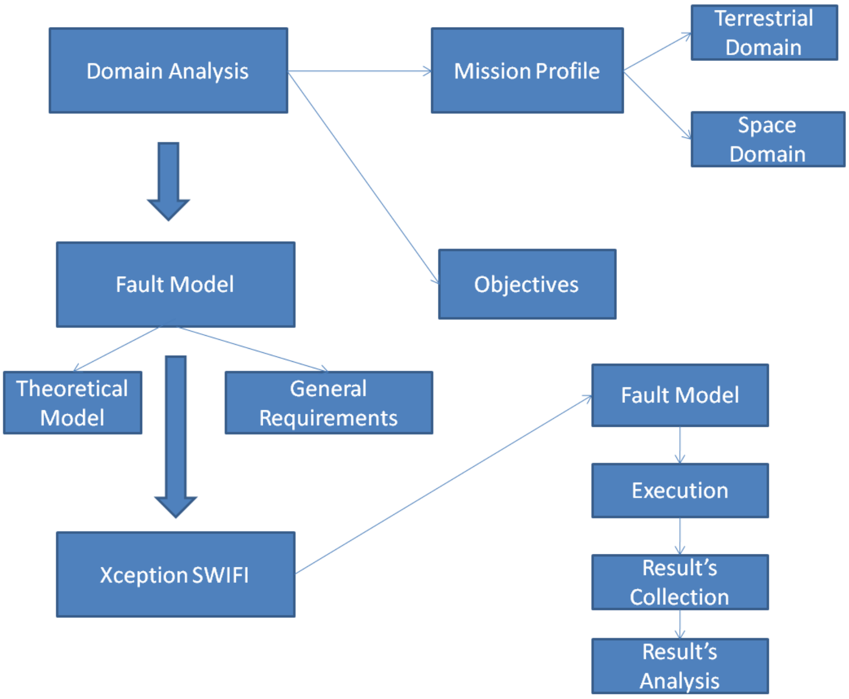


Figure 8 - SWIFI Methodology main items

Domain Analysis

Taking into account that the primary goal is to inject faults (and retrieve the subsequent results) in a microprocessor to be used in spacecrafts, one should point out the importance of this application. Due to its lower price and faster deployment, space mission critical project managers resort to commercial-off-the-shelf (COTS) components in order to build their systems.

This raises one very important question: if specific materials/components are prone to suffer from fatal errors, what about the less reliable COTS components (which are not hardened or built specifically for critical missions)?

It is often neither possible nor cost effective to build systems without using these components, therefore there is only one way to answer this question: to evaluate the dependability of these systems. Before analysing how can these systems be evaluated in an efficient way one should consider the cause of "real world" faults in adverse environments.

Damage due to radiation (in space) can be divided into two groups, the total ionizing dose (TID) and single event effects (SEE). The former is a cumulative degradation that takes place in the long term while the latter can be described as charged particles that change the state of the microcircuits. The SEE group comprehends a wide variety of events. Three of the most important are the Single Event Upset (SEU) which usually results in bit-flips, the Single Hard Error (SHE) which cause a permanent change to the operation of a device (usually a permanent stuck bit in a memory device) and a Single Event Functional Interrupt (SEFI) which leads to a state in which the device stops functioning correctly, and requires a reset in order to resume normal behaviour [NASA09]. All these effects can, obviously, have dire consequences and jeopardize a costly and important mission.

The SEE can also be present at a ground level, it is known that microcircuits had been affected and led to dozens of servers crashing due to bit-flips caused by cosmic rays as well as sensitive electronic devices were used near high voltage devices, such as trains and industrial appliances [Normand04]. Similar effects have happened in avionics. Due to its difficult detection, incidents were observed by pilots until it was proven scientifically [Normand04]. Following a series of proprietary flight experiments by IBM on aircraft in which upsets were measured lead to a complete and thorough study that proves the existence and commonness of these phenomena in avionics [James92].

Considering all the problems that arise from critical system's malfunctioning and the aforementioned techniques, one must ask "what are the objectives of fault injection?". This question should be divided into two. These are: "How can one say that the dependability of a system is high?" and "How can one trust the test procedures used during the development of the system?". The fault injection technique can verify if the implemented fault tolerance mechanisms function in a proper way and if not it helps to re-design the system in order to remove the presence of faults. In reply to the former question one can say that the fault injection technique can also help determine the efficiency of test procedures undertaken during the development phase [Arlat93].

More precisely, fault injection techniques provide a way for fault removal and fault forecasting which yields three benefits:

- The ability to understand the effects of real faults and of the related behaviour of the system;
- Understand and assess the efficacy and efficiency of the fault tolerance mechanisms existent in the system being tested.
- Forecasting the target system's faulty behaviour [Benso03].

Xception SWIFI Fault Model and General Requirements

The Porting Methodology clearly states that the plug-in should be contextualised scientifically by comparing it to the FARM model (4.2.1). The F set comprehends the fault location (see 3.3.1), the fault triggers (temporal or instruction fetch, see 3.3.1) and the fault types (bit flip, see

3.3.1). As explained in the domain analysis this makes possible for the Xception tool to fulfil the representativeness general requirement.

The A set is not particularly important for this plug-in because any program can be used and since faults can be injected in both user and kernel space, all code running in the system can be considered for the activation set.

The R set is made of a group of classifications depending if the fault injection was detected, if the program terminated correctly or not and if the final result was equal to a run without fault injection (gold run).

The M set is composed of valuable information about condition the registers are in (before and after a fault injection), the F set used and the outcome of the fault injection (how the system handled it). These features allow this plug-in to respect the inspection general requirement.

The Xception application has a one sided branch tree approach for organising the fault injection process. The root of that tree is the Campaign. This element is a logical entity that contains a set of Experiments (the second node of the tree) usually defined by having the same evaluation goal which is, subsequently, composed by a set of Injection Runs (the third node of the tree). This last node is composed by a sequence of fault injections.

Execution is explained by the figure 9.

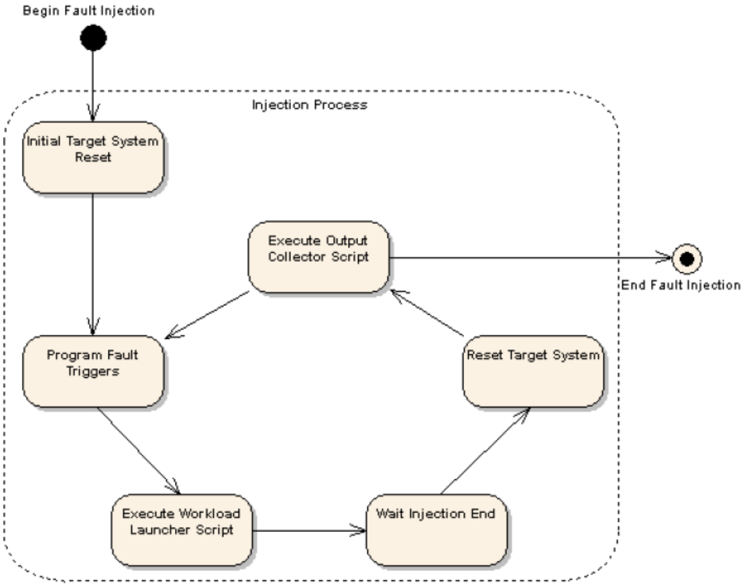


Figure 9 - Xception SWIFI plug-in general execution

This execution is made with a very small memory usage, below 100 kbytes, enabling the Xception tool with a very low memory footprint, thus respecting the intrusiveness general requirement.

Collecting results is essential to assess the system's response to fault injections. The Xception application has the Xtract add-on tool that provides a set of pre-defined queries which act upon the Xception database. The outcome of these queries provides the user with an easy to understand layout.

This tool allows the user to browse the aforementioned four main entities in the Xception database: Campaigns, Experiments, Injection Runs and Faults. This way the user can identify everything that was run, the faults characteristics and the output, i.e. the registers' state, workload output and if the fault was injected.

It is of paramount importance to analyse the outcome of the fault injections by having a Fault Detection Isolation and Recovery (FDIR) perspective. First, one should first look at the fault, error, failure chain on figure 10.

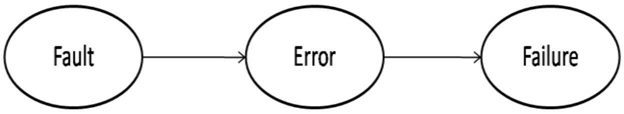


Figure 10 - Fault, Error, Failure chain

A fault is a physical defect, imperfection, or flaw that occurs within some hardware or software component. An error is a deviation that reflects the fault. The failure is the inability of a system to perform its required functions within specified performance requirements. When a fault causes an incorrect change an error occurs. When the fault tolerance mechanisms detect a fault/error they try to handle and contain them. If it is not able to do that, it will eventually malfunction and a failure occurs. [Benso03]

The FDIR falls into two categories:

- How the mechanisms handle the errors?;
- The outcome.

The following state machine, depicted in figure 11 shows how a system can detect (or not) a fault and how it handles it. The outcome will have a classification which will subsequently be used for a statistical analysis by the Xception tool.

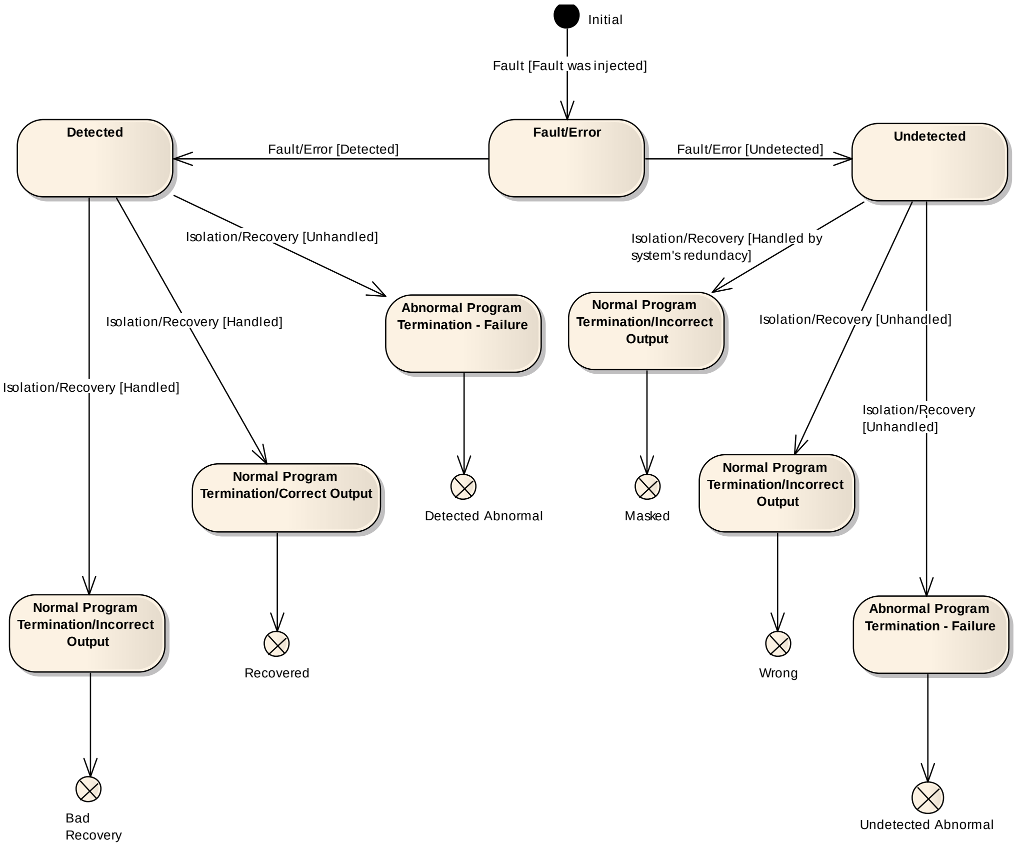

Initial
Fault [Fault was injected]
Detected
Fault/Error [Detected]
Fault/Error
Fault/Error [Undetected]
Undetected
Isolation/Recovery [Unhandled]
Isolation/Recovery [Handled by system's redundacy]
Isolation/Recovery [Handled]
Abnormal Program Termination - Failure
Normal Program Termination/Incorrect Output
Isolation/Recovery [Unhandled]
Isolation/Recovery [Handled]
Isolation/Recovery [Unhandled]
Normal Program Termination/Correct Output
Detected Abnormal
Masked
Normal Program Termination/Incorrect Output
Normal Program Termination/Incorrect Output
Abnormal Program Termination - Failure
Recovered
Wrong
Bad Recovery
Undetected Abnormal

Figure 11- Outcome Classification State Machine

This analysis allows the Xception tool to respect the general requirements of observability and of inspection.

5.1.2 Original and Destination Hardware User Manual and Programming Manual and Operating System

An extensive study was performed in order to fully understand the differences between the PowerPC 603e and the PowerPC 750 Microprocessor. The programming manual was consulted in order to understand the ground rules to program these two microprocessors. Each User Manual was carefully studied so that the differences could be reported and afterwards determine which differences might have an impact on the implementation phase. The conclusions of this report are straightforward. The fault locations are present and function in the same way, although the PowerPC 750 has two Integer Units [IBM99]. The differences lie in terms of features, i.e., faster calculations, clock rate, cache, performance and thermal control. The PowerPC 603e uses two instructions only common to this Microprocessor and the PowerPC 750 has 49 more instructions than the PowerPC603e, see Annex A, A.2.

Regarding registers there are several differences. These difference lie in three major groups:

- Existence/non-existence.
- Different levels of access to a set of registers.
- Differences in the bits that comprehend a set of registers.

Regarding the first point, it can be said that the PowerPC 603e has the Software Table Search Registers that cannot be found in the PowerPC 750. On the other hand the PowerPC 750 has unique registers, Performance and Power Monitor registers as well as the L2 Cache Control register [IBM99].

Regarding the second point, one can say that a group of registers common to both microprocessors are located in different layers of access. After studying the OS, these differences become unimportant because it is able to access all layers.

Regarding the final point, it can be said that the Machine State Register (MSR) and the Instruction Address Access Register (IABR) are somewhat different in terms of the functionalities it provides by manipulating its bits.

For more details about these differences and the full comparative study, consult Annex A.

5.1.3 Template Documents, SRS, SAS, DBS, TCS

Due to lack of documentation regarding the PPC603e plug-in, it was necessary to produce all documentation without resorting to already written documents. In terms of requirements, it was necessary to specify how the plug-in should handle the fault injection process and all possible mishaps, more importantly it was defined the collection of new data, the way the Communication Proxy (EME-Proxy) should handle the sequence of fault injections and the type and structure of messages to be exchanged in order to guarantee a seamless fault injection process. This was only possible because this phase was overlapped with the Design Phase.

The following state diagram (figure 12), was established for the EME-proxy. Please notice that in order to maintain readability and decrease redundancy, a group of transitions were omitted. These transitions are to the Error state. These transitions occur whenever the state is not defined, the state is defined but not in the compound state it is in, if the state is defined and it is in the Injection Stage compound state (the EME-Proxy should not be running during injection), and when the received message (from the host) size equals 0. One should take into account that every time the EME-proxy sends a resetting message the EME-proxy closes. This case is only shown when there is a transition to the Error state and subsequently when it does not receive a ResetRequest message.

For instance, in the compound ReceiveFaultDefinition State the EME-proxy starts the fault injection process, it resets and when the fault injection ends it will come back and start at the InjectionEnd Stage compound state. One of the particularities of this state machine is the fact that it does not have an exit when no errors are encountered. This is so because it is the EME, in the host computer, that decides when to finish the fault injection process.

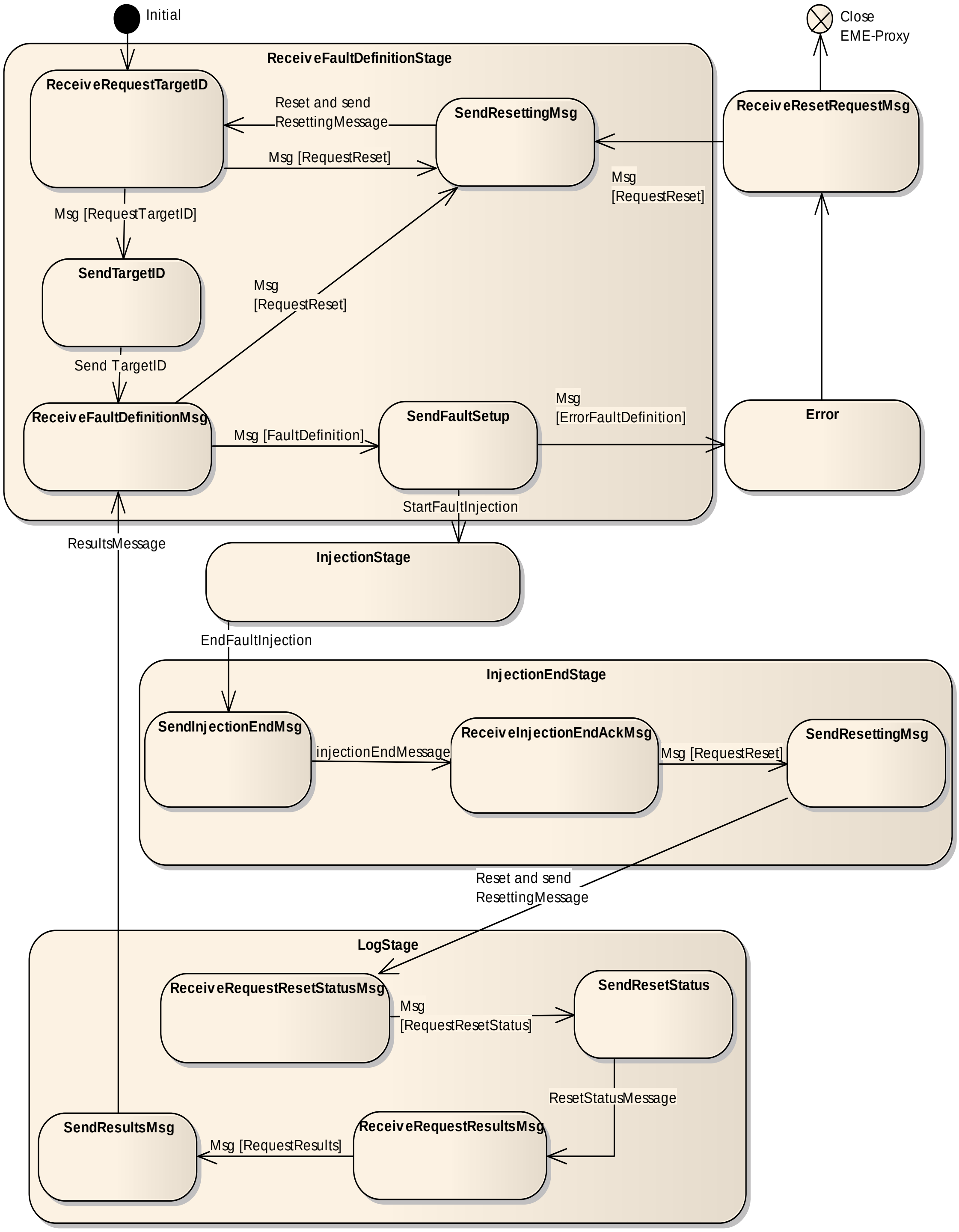


Figure 12 - EME-Proxy State Machine

In terms of architecture, the original architecture was studied and the architecture for the new plug-in was defined. Once again, this was only possible due to overlapping with the Design Phase. The figure 13 depicts a package overview of the plug-in. One should notice that the Infobus is a package that provides a set of methods that allow the exchange of data asynchronously between the several components of the plug-in in the Host computer.

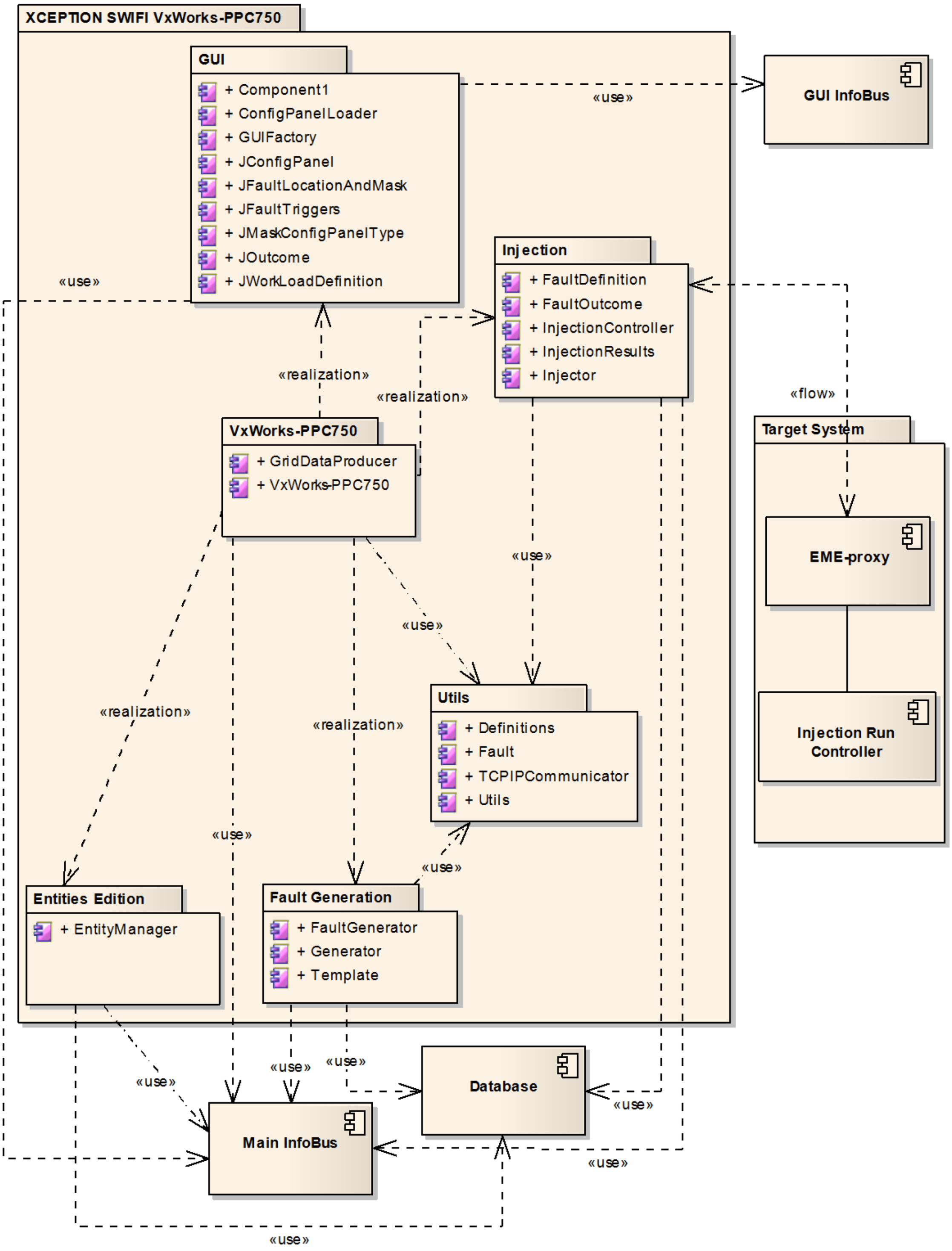


Figure 13 - Plug-in architectural overview

These packages are:

- VxWorks-PPC750 – Responsible for initialising the plug-in.
- Injection – Responsible for the fault injection process, i.e. communicating with EME-proxy, sending fault definition, collecting results.
- Fault Generation – Responsible for the generation of fault parameters.
- GUI – Responsible for the plug-in's graphical interfaces.
- Entities Edition – Responsible for the editing operations.
- Utils – Contains definitions and tools necessary for communication between the components.
- Target System – Responsible for the actual fault injection at the microprocessor level.

Each package has a set of components that have specific functions that are explained in the following sections.

VxWorks-PPC750

This package is described in the figure 14 and has the main plug-in's component. The main necessary objects are instantiated here.

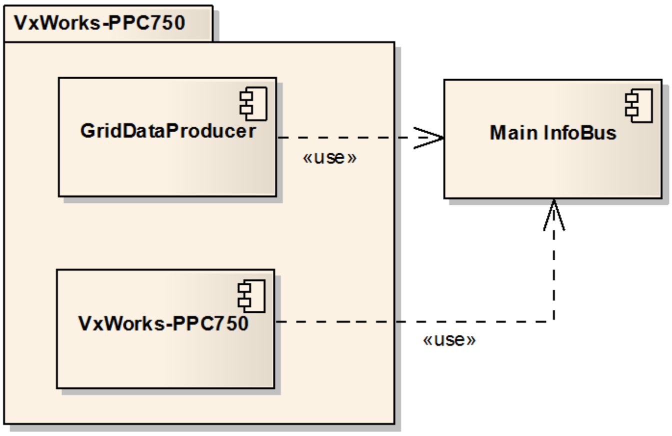


Figure 14 - VxWorks-PPC750 package

The component responsible for these initializations is the VxWorks-PPC750. Adds the Plug-in to the main application (EME) and makes sure it can properly communicate with it. Instantiates objects related to the Entity Manager, injection, GUI and fault generation. The GridDataProducer component is responsible for adding information to the EME main window. Both these components use the Main infobus in order to communicate with the whole application.

Injection

This package is responsible for the fault injection process and is described in the figure 15.

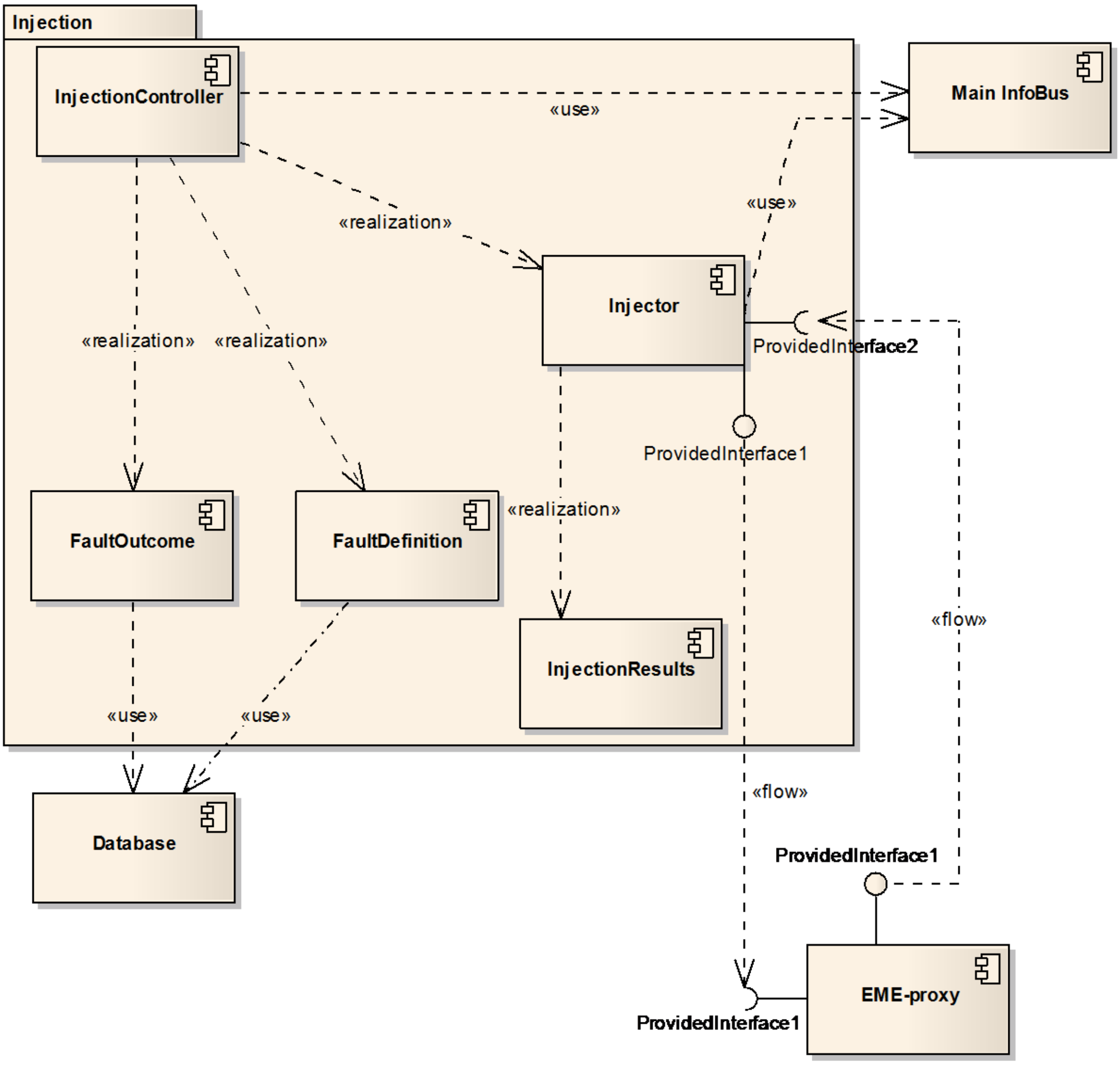


Figure 15 - Injection Package

The injection controller is the main component of this package. It receives fault definitions and creates injectors as well as fault outcome components. Each injector is responsible for establishing communication with the EME-proxy, send fault definitions and gather results, which in turn are processed by the InjectionResults component. The FaultOutcome package is responsible for storing the results in the database. The FaultDefinition is responsible for building the fault definition that is to be sent to the target. In order to do that it accesses the database. The InjectionResults package is responsible for sending the injection run results and for sending it to the EME. Both the InjectorController and the Injector use the Main Infobus component to communicate with the whole application.

Fault Generation

This package is responsible for the generation of fault parameters and to store them in the database.

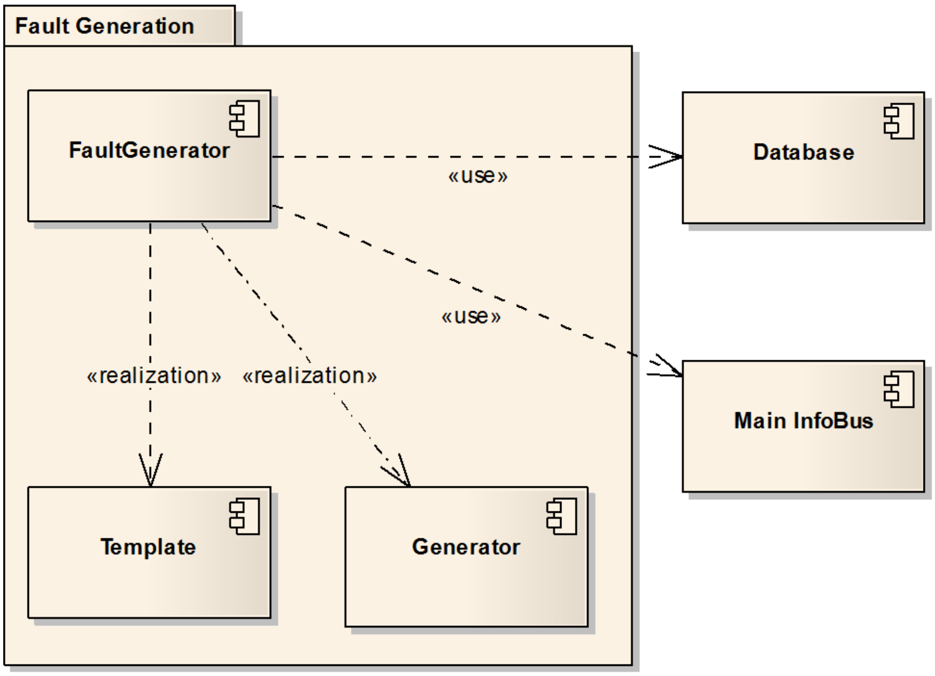


Figure 16 - Fault Generation Package

As described by the figure 16 it receives, via Main Infobus, the fault generation parameters defined by the user. The main component for these actions is the FaultGenerator. This component uses the Template for building the data that is to be inserted in the in the database when the option New Template is chosen. It also uses the Generator to create fault parameters.

GUI

This package is responsible for everything that relates to the Plug-in's graphical interface and is described in figure 17.

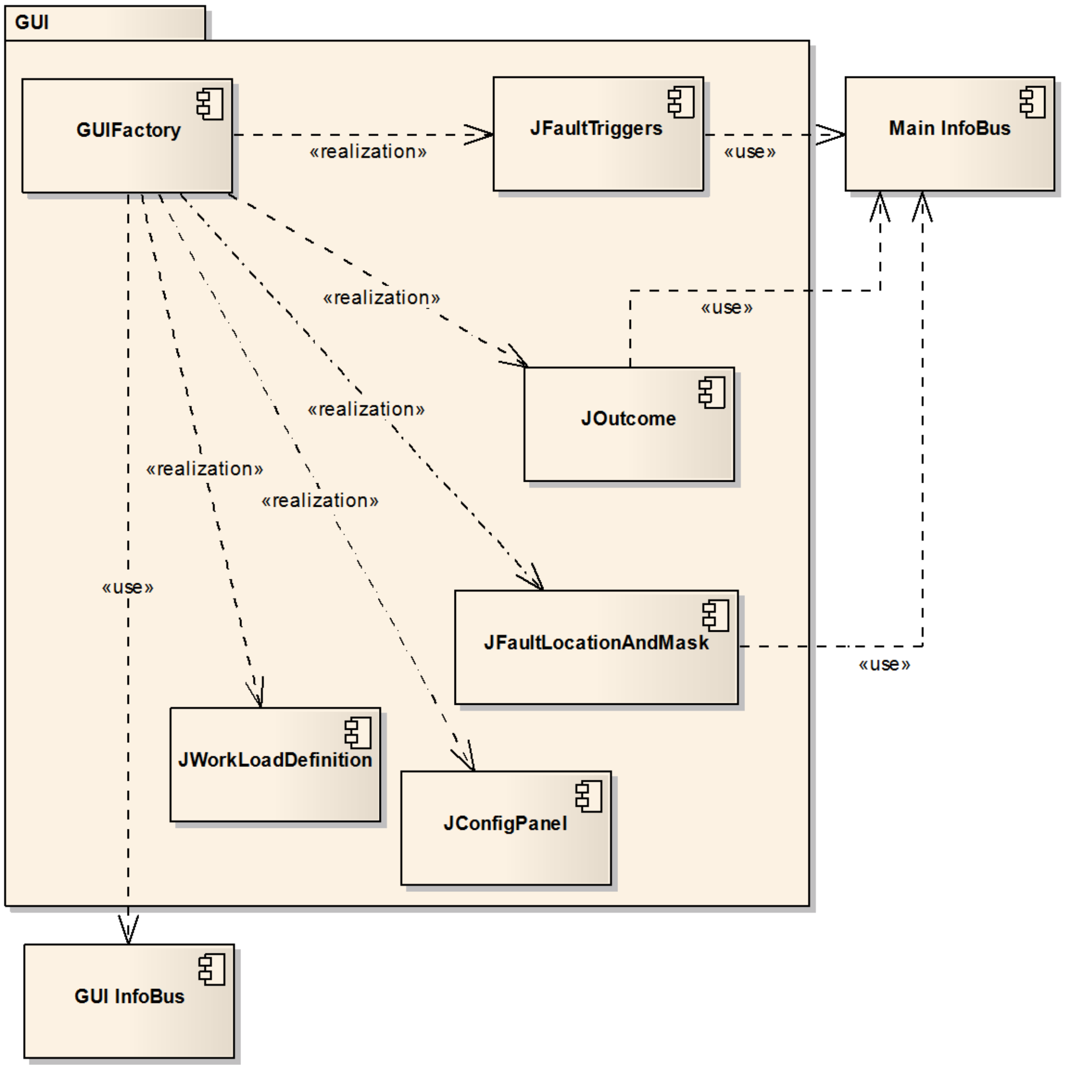


Figure 17 - GUI Package

The GUIfactory component is responsible for providing all panels and dialogs and it uses the GUI infobus to know which panels/dialogs are requested. The JFaultTriggers component is responsible for the fault triggers configuration panel. The JOutcome is responsible for the outcome configuration panel, The JFaultLocationAndMask is responsible for the fault location and mask definition panel. The JworloadDefinition is responsible for the panel with the workloads description. The JconfigPanel is responsible for the Plug-in configuration panel.

Entities Edition

This package, described in figure 18, and its only component is responsible for processing four database operations available in the EME data grid. These are Delete, Copy and Paste of entities, i.e., Campaigns, Experiments, Injection Runs, Faults.

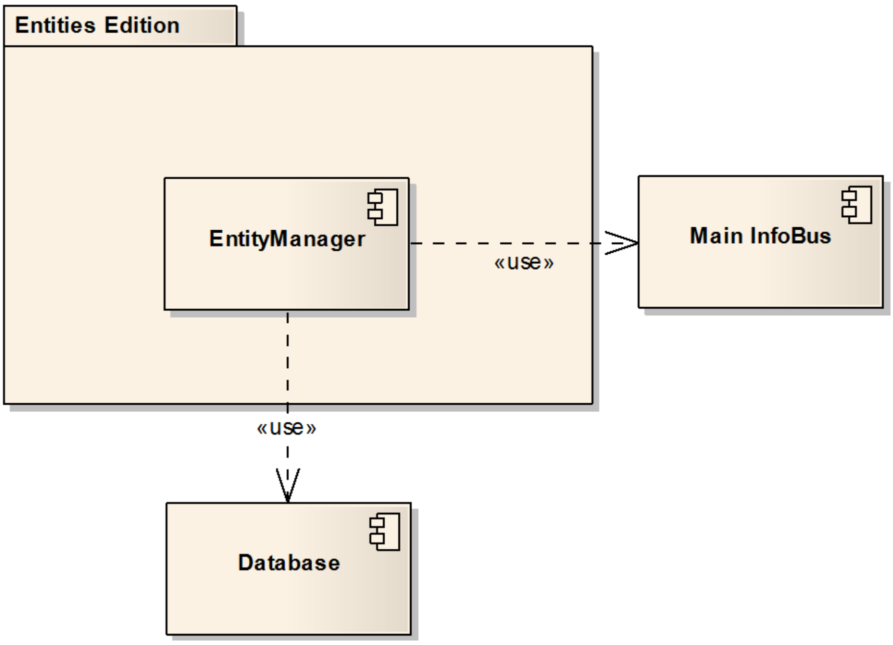


Figure 18- Entities Edition Package

Utils

This package is responsible for miscellaneous actions and definitions and is described in figure 19.

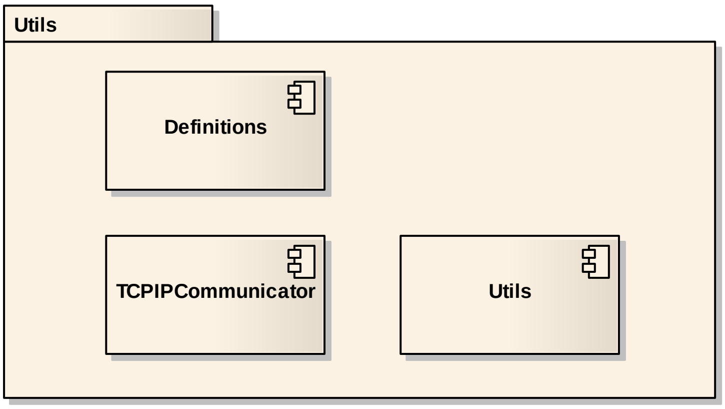


Figure 19 - Utils Package

The Defintions package holds global definitions used throughout the plug-in. The TCP-IP Communicator is responsible for the implementation of methods that allow the establishment of communication with the target system. The Utils package holds a set of miscellaneous methods that are not suitable, in terms of modelling, to be held by other components.

Target System

This package is responsible for the actual fault injection at the Microprocessor and is described in figure 20.

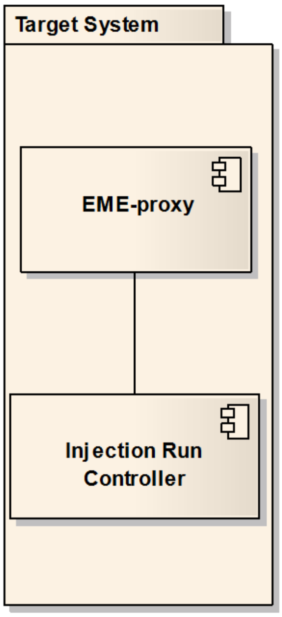


Figure 20 - Target System Package

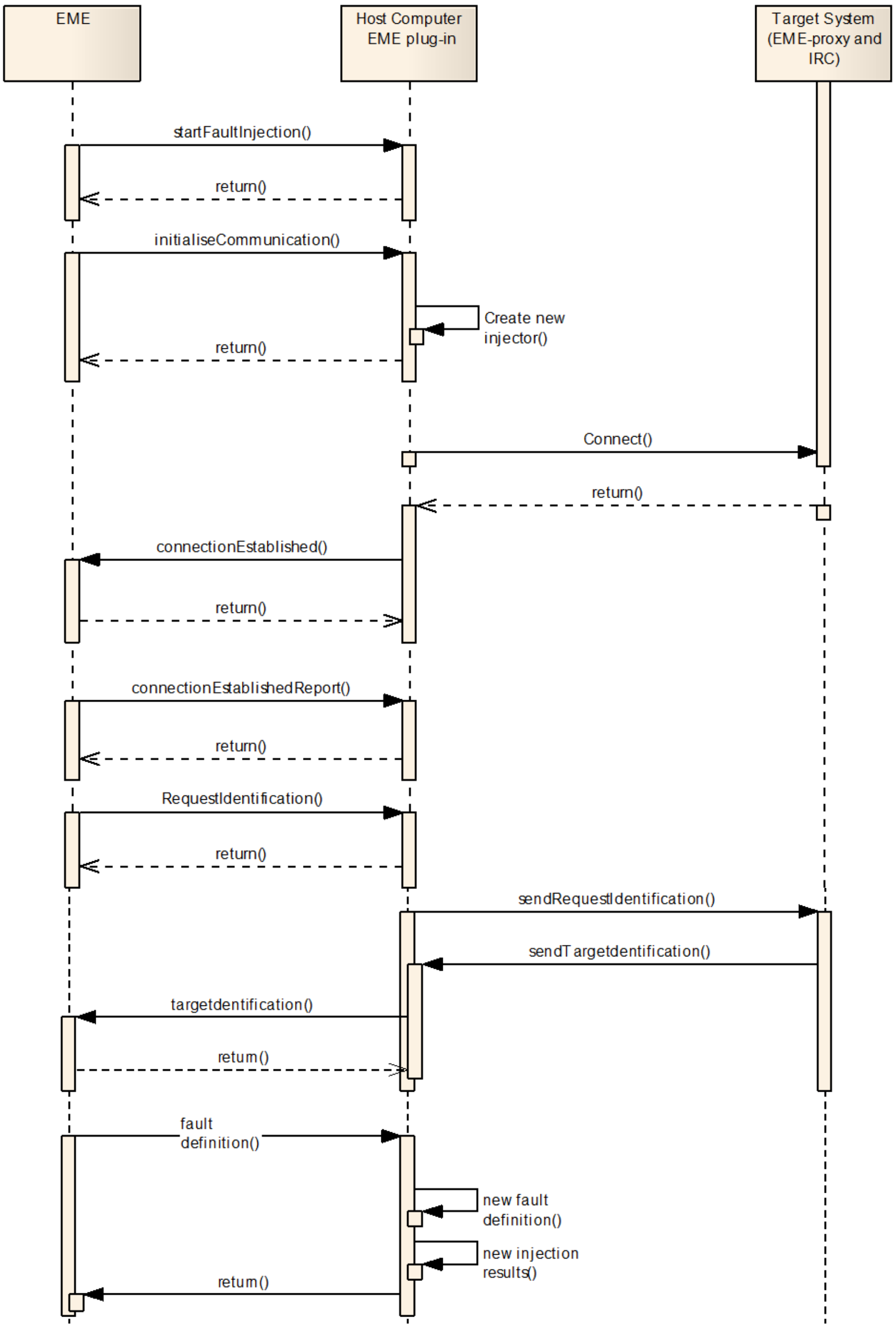
The EME-proxy is responsible to receive all the necessary instructions to perform fault injections, i.e., fault definition. With this information it resorts to the Injection Run Controller to inject the fault and collect the results. Afterwards, it sends the results to the Host computer.

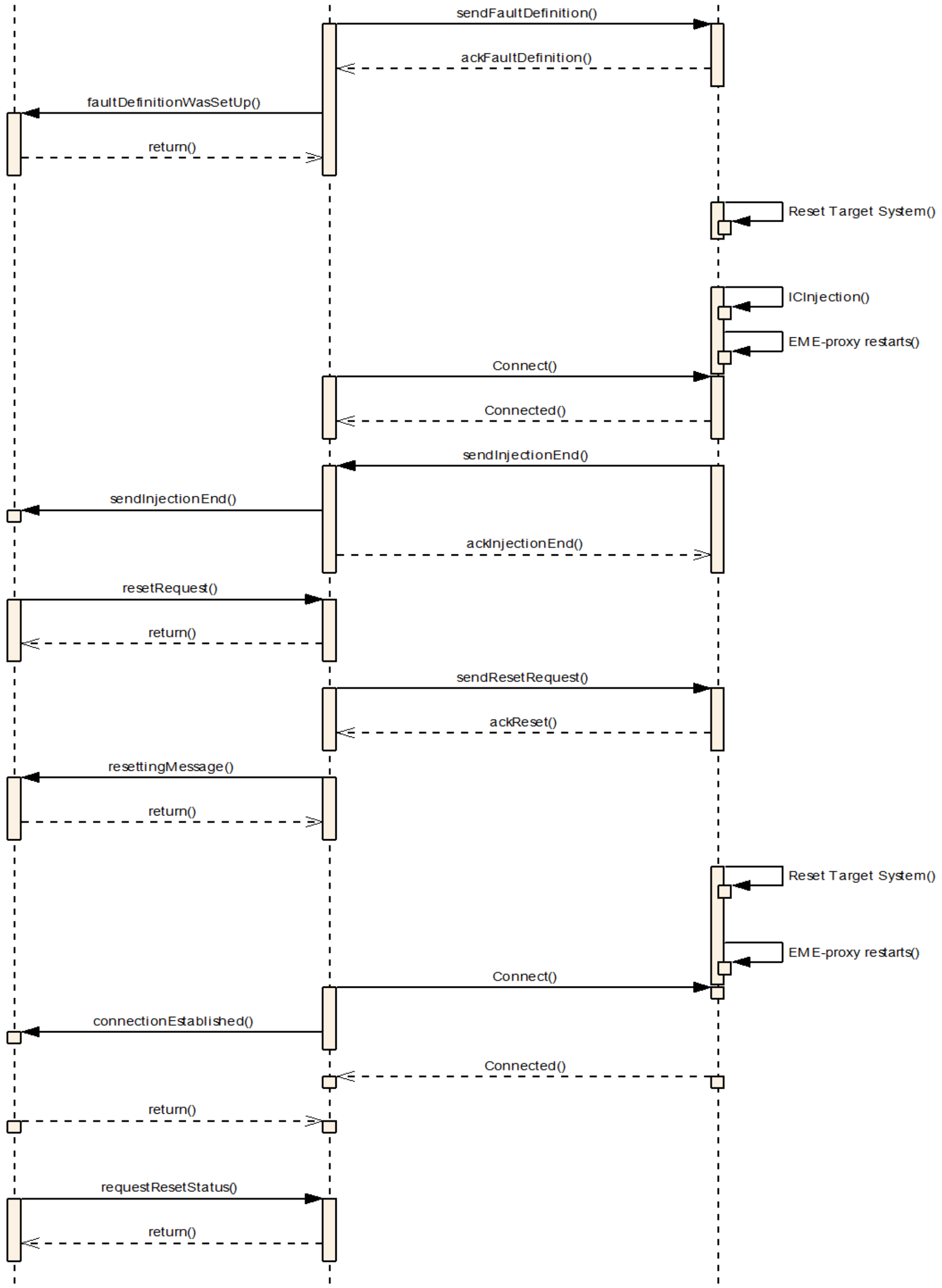
It was developed a test case specification document in order to allow for a full fledged testing procedure after the implementation is complete. These tests were related to all the operations with which it is possible to operate on the plug-in as well as underlying communication between the several components and the actual fault injection.

In terms of database specification, it was already specified. The structure of the tables remains the same, only a change to the outcome results columns, because of the registers differences.

5.2 Design

The PPC603e plug-in design was analysed by resorting to the documentation tool Doxygen. This allowed to get acquainted with the subtleties of the plug-in and to synchronise the existing and usable code with the previously developed artefacts, as well as to produce a clearly defined design for the PowerPC 750 plug-in. As mentioned before, this phase helped define the previously described architecture. The other main output of this phase was the definition of sequence that the whole fault injection process should follow. This sequence can be seen in figure 21.


EME
Host Computer EME plug-in
Target System (EME-proxy and IRC)
startFaultInjection()
return()
initialiseCommunication()
Create new injector()
return()
Connect()
return()
connectionEstablished()
return()
connectionEstablishedReport()
return()
RequestIdentification()
return()
sendRequestIdentification()
sendTargetdentification()
targetdentification()
return()
fault definition()
new fault definition()
new injection results()
return()


sendFaultDefinition()
ackFaultDefinition()
faultDefinitionWasSetUp()
return()
Reset Target System()
ICInjection()
EME-proxy restarts()
Connect()
Connected()
sendInjectionEnd()
sendInjectionEnd()
ackInjectionEnd()
resetRequest()
return()
sendResetRequest()
ackReset()
resettingMessage()
return()
Reset Target System()
EME-proxy restarts()
Connect()
connectionEstablished()
Connected()
return()
requestResetStatus()
return()

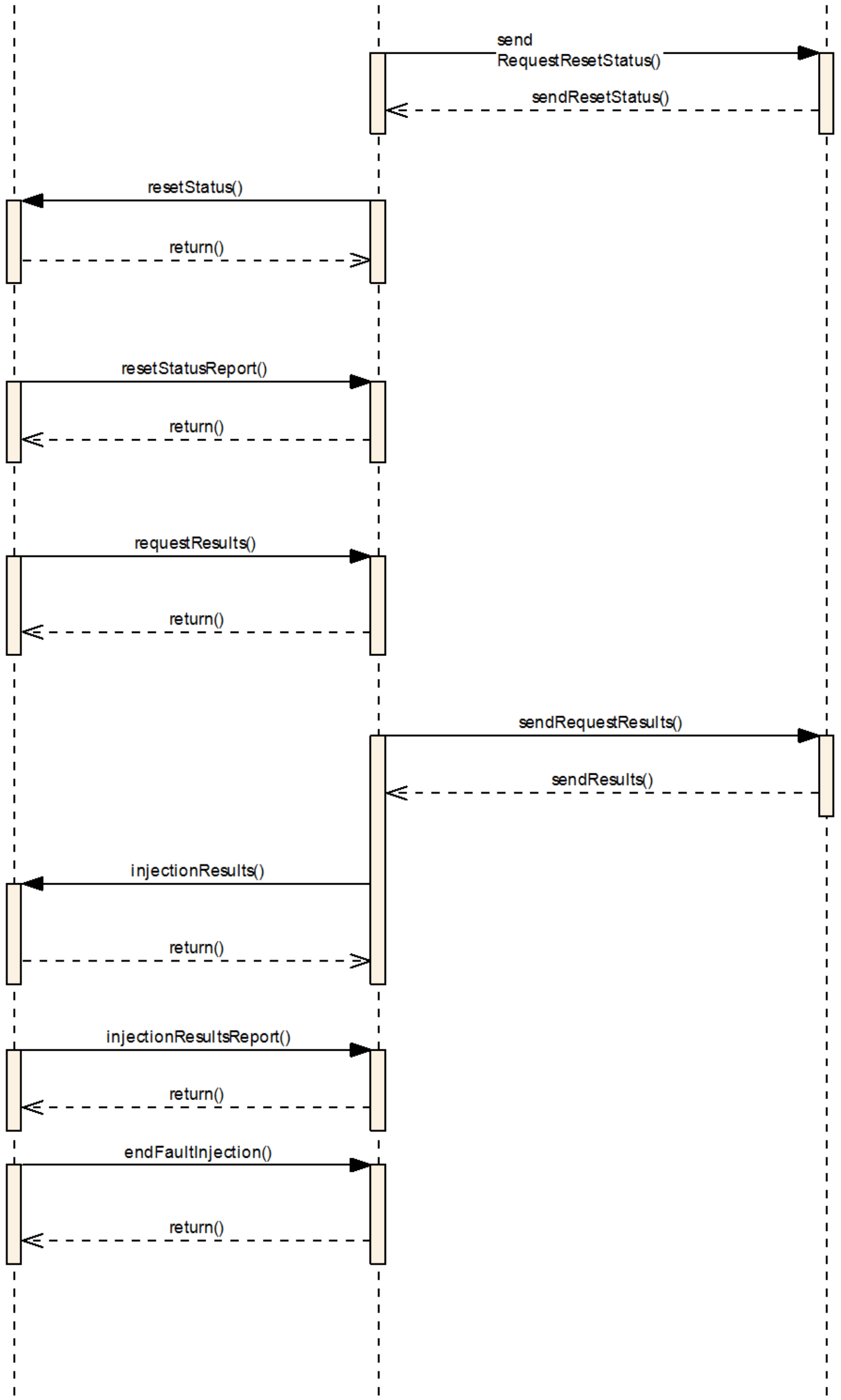


Figure 21 - Sequence Diagram of Fault Injection Process

There are very important details that should be taken into account regarding the sequence diagram. These are:

- The need for a proper connection at the beginning of the fault injection process.
- The ability to reconnect to the Target System, otherwise, the fault injection must be aborted.
- The ability to properly reset the target system without delaying the whole fault injection process.

- Proper processing and communication of the fault definition.
- Proper processing and communication of the fault injection results.

5.3 Implementation

Taking into consideration that the PowerPC 750 Microprocessor has a different register set the first measure was to alter the gathering of information, at assembly level, in fault injection process about the non-present PowerPC 603e Software Table Search Registers and adding the Performance and Power Monitor set of registers as well as the L2 Cache Control register.

Since the amount of information to gather increased, changes had to be done in the amount of data allocated in dynamic memory. This had to be done with special care because of the general requirement concerning intrusiveness, i.e., the application running on the target system should have the least possible footprint, in terms of processing as well as memory. The total amount of memory necessary reached 65550 bytes, below the maximum allowed by the defined requirements – 100000 bytes.

In order to inject faults determined by temporal triggers, changes had to be performed in the implemented algorithm in order to reflect the faster clock rate of the PowerPC 750 Microprocessor.

The Instruction Address Breakpoint Register, which allows the software to be acquainted of the next instructions to be executed, is essential for injection faults determined by spatial triggers. Due to the differences between the two microprocessors, modifications had to be performed in order to change how it is accessed and used.

As mentioned before, in order to inject faults in the Floating-point Unit and the Integer Unit a trigger is chosen and when it is activated a search is performed for instructions used by the chosen location. Taking this into account, new instructions had to be added and the search algorithm had to be changed. This change resulted from the fact that the search is performed based on the opcode of each instruction and these new instructions have completely different opcode structure.

Due to more data being collected than the PowerPC 603e version, the Communication Proxy had to be changed in order to sustain the increase output. For this reason, changes had to be made in component running in Host computer in order to support this new amount of input.

5.4 Acceptance

Thorough tests have been performed in order to assess that the port was successful. Besides testing how the plug-in performs in an operation point of view, special attention was given to very specific aspects. These were:

- Check if the fault injection algorithms perform correctly, more specifically the ones that resort for the altered search algorithm.
- Check if the data collecting, at assembly level, is being carried out in a correct way.
- Check if the Communication Proxy is handling correctly a bigger amount of information.

- Check if the exchange of messages was correctly implemented and if it provided a bug-free fault injection process.
- Check if the Host computer handles correctly the new amount of input of information.

In order to perform these verifications unit testing was performed in the Host computer plug-in and all possible fault injections were tested at least once in the target system.

5.5 Operations and Maintenance

No maintenance plan was developed nor any feasibility report. This is due to the fact that these documents should only be written when there is an interest from the client.

5.6 Summary

This chapter describes the porting process based on the previously defined porting methodology.

In the Requirements Elicitation phase, all requirements pertaining the plug-in were defined with special emphasis on the communication process between the target system and the host computer and the fault injection process.

In the Design phase it was defined the sequence of the fault injection process.

In the Implementation phase the necessary changes were made in order to port the plug-in for the PowerPC 750 Microprocessor. These changes were related to the results collection, temporal triggers and the algorithm for searching instructions that is the cornerstone for injecting faults in the Floating-point Unit and Integer Unit.

In testing phase tests were performed in order to establish if the plug-in is working correctly.

The operations and maintenance was not executed.

Chapter 6

Conclusions and Future Work

Don't be too proud of this technological terror you've constructed. The ability to destroy a planet is insignificant next to the power of the Force.
Darth Vader

This chapter presents a summary of the research work and its outputs. Conclusions are drawn about the experimental results and contributions of the project are described. Suggestions for future work are made.

6.1 Contributions

The main contributions of this research were:

- Definition of a Porting Methodology which purpose is to ease the porting process of Xception tool SWIFI plug-ins.
- Contextualising scientifically the Xception SWIFI plug-in.
- Porting the VxWorks-PPC603 plug-in to the PowerPC 750 Microprocessor.

By using the defined porting methodology it was possible to port the VxWorks-PPC603 plug-in. What seemed to be a complex project at the beginning became a clearly defined process that provided with timely results at each phase and in the end a successful port.

In terms of experimental results it can be said that the resulting plug-in kept all the general features necessary to perform fault injections. This means that the necessary changes to the internal structure were made without any changes in its observable behaviour. Due to the differences between both Microprocessors, new functionalities were added in order to use the PowerPC 750 characteristics. These functionalities were fully implemented and have a correct behaviour.

6.2 Future Work

Regarding future work it should be noticed that new templates, based on the already produced documentation, should be produced in order to ease the porting process in the future.

This porting methodology should be applied in more case studies in order to assess how generic it is and its applicability. In this project the User Manual and Installation Manual were not produced, however the lack of these documents had no implications in the evaluation of the proposed methodology.

Bibliography

[Benso99] A. Benso, M. Rebaudengo, M.S. Rorda. "Fault Injection for Embedded Microprocessor-based Systems." Journal of Universal Computer Science, 1999.

[Benso03] Alfredo Benso, Paolo Prinetto. Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation. Kluwer Academic Publishers, 2003.

[Baldini00] Andrea Baldini, Alfredo Benso and Paolo Prinetto. ""Bond": An Agents-Based Fault Injector for Windows NT." *IEEE,* 2000

[Boehm09] Boehm, B. "Managing Software Productivity and Reuse." *IEEE Computer*, 1999.

[CSW03] Critical Software, S.A. "Xception." 2003. www.xception.org (accessed 06 11, 2009).

[Caignet01] F. Caignet, S. Delmas-Bendhia, E. Sicard. "The challenge of signal integrity on deep-submicron CMOS technology." *IEEE*, 2001.

[Kanawati92] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob, A. Abraham. "FERRARI: A Tool for The Validation of System Dependability Properties." *IEEE*, 1992.

[IBM99_A] IBM. MPC603ee RISC Microprocessor User’s Manual.1999.

[IBM99] IBM. PowerPC 750 PowerPC RISC Microprocessor User’s Manual. 1999.

[IBM00] IBM. PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors. 2000.

[IBM05] IBM. PowerPC® Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors.2005.

[Arlat93] J.Arlat, A.Costes, Y. Crouzet, J.C. Laprie, D. Powelll. "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems." *IEEE Transactions on Computers*, 1993.

[Arlat02] Jean Arlat, Jean-Charles Fabre, Manuel Rodriguez, Frédéric Sales. "Dependability of COTS Microkernel-Based Systems." *IEEE*, 2002.

[Arlat90] Jean Arlat, Martine Aguera, Louis Amat, Yves Crouzet, Jean-Charles Fabre, Jean-[Laprie90] Claude Laprie, Eliane Martins, David Powell. "Fault Injection for Dependability Validation: A Methodology and Some Applications." *IEEE*, 1990.

[Johnson89] Johnson, W. B. *Design and Analysis of Fault Tolerance Digital.* Addison-Wesley Publishing Company, 1989.

[Laprie95] Laprie, Jean-Claude. "Dependability of Computer Systems: Concepts, Limits, Improvements ." *IEEE*, 1995.

[Leheym95] Leheym, Greg. *Porting UNIX Software.* O'Reilly Media, 1995.

[Schuette90] M. Schuette, J. Shen, D. Siewiorek, Y. Zhu. "Experimental evaluation of two concurrent error detecting schemes." *IEEE*, 1990.

[Rodriguez02] Manuel Rodriguez, Arnaud Albinet, Jean Arlat. "MAFALDA-RT: A tool for Dependability Assessement of Real-Time Systems." *IEEE*, 2002.

[McHale04] McHale, John. *Commercial aircraft avionics provide technology basis for future space exploration vehicle.* 2004. http://www.optoiq.com/articles/display/349922/s-articles/s-military-aerospace-electronics/s-exclusive-content/s-commercial-aircraft-avionics-provide-technology-basis-for-future-space-exploration-vehicle.html (accessed 06 11, 2009).

[NASA03] NASA. *NASA Awards NENS Contrat to Honeywell Technology Solution.* 10 2003.http://www.nasa.gov/home/hqnews/2003/oct/HQ_c03jj_near_earth_network.html (accessed 06 11, 2009).

[NASA00] NASA. *NASA/GSFC Radiation Effects and Analysis home page.* http://radhome.gsfc.nasa.gov (accessed 04 14, 2009).

[Normand04] Normand, Eugene. "Single Event Effects in Avionics and on the Ground." *International Journal of High Speed Electronics and Systems*, 2004.

[Fouillat04] P. Fouillat, V. Pouget, D. Lewis, S. Buchner, D. McMorrow. "Investigation of Single-Event Transients in Fast Integrated Circuits with a Pulsed Laser." *International journal of high speed electronics and systems* , 2004.

[Velazco 02] R. Velazco, A. Corominas. " Injecting Bit Flip Faults by Means of a Purely Software Approach :a case studied." *IEEE*, 2002.

[Sommerville07] Sommerville, Ian. Software Engineering 8. International Computer Science Series, 2007.

[Vinson92] Vinson, James Edwin. "Circuit Reliability of Memory Cells ." *IEEE*, 1992.

[Voas99] Voas, J.M. "Certifying Off-the-Shelf Software Components." *IEEE*, 1999.

[Carreira98] J.Carreira, H.Madeira, J.G. Silva. "Xception: A Technique for the experimental evaluation of dependability in modern computers." *IEEE,* 1998.

[Dasilva07] Antonio Dasilva, José-F Martínez, Lourdes López, Ana-B Garcia and Luis Redondo. "Exhaustif: A fault injection tool for distributed heterogeneous embedded systems ." *Euro American Conference On Telematics And Information Systems*, 2007.

Glossary

EME-proxy Component of the VxWorks-PPC603 and VxWorks-PPC750 plug-in for the Xception tool responsible to communicate with the Host computer in order to perform the fault injection process.

Fault Definition Message that describes the fault to be injected. It is composed of fault location and trigger.

Host computer It describes the computer connected to the target system in the Xception tool. It runs the application used by the user to define faults and to initiate the fault injection process. This application is also responsible to save all the data pertaining the fault injection process.

Injection Run Controller Component of the VxWorks-PPC603 and VxWorks-PPC750 plug-in for the Xception tool responsible for performing the fault injection and collect its results.

Software Development Process Abstract representation of a software process that is composed of a set of activities that leads to the production of software.

SWIFI It stands for Software Implemented Fault Injection and is one of the techniques for injecting faults into a system resorting purely to software.

SWIFI methodology Describes the general concepts of a SWIFI validation tool. It defines fault locations, triggers, collection of data (results) and its analysis.

Target System System where it is located the Microprocessor where Xception SWIFI plug-in injects faults.

Annex A

Comparative study of the PowerPC 603e and PowerPC 750 Microprocessors

A.1 PowerPC 603e/750 Architecture

In this section a general overview of the PowerPC 603e/750 architecture will be presented.

A1.1 Overview

The PowerPC 603e/750 microprocessor is an implementation of the 32 bit PowerPC family of RISC microprocessors. These types of processors use 32 bit effective addresses, integer data types of 8, 16 and 32 bits and floating-point data types of 32 and 64 bits [IBM99][IBM99_A].

It's a superscalar processor, which means that it can execute more than one instruction during a clock cycle. This is done by using several functional units that are on the processor. The execution units used are an integer unit (IU), a floating-point unit (FPU), a load and store unit (LSU), system register unit (SRU) and a completion unit (CU). The instructions units used are the instruction queue (IQ) and the branch processing unit (BPU) [IBM99][IBM99_A].

Relevant execution/instruction units and memory subsystem

The BPU receives branch instructions and performs lookahead operations on conditional branches in order to resolve them early and (hopefully) achieve a zero-cycle branch. This unit uses three-user control registers. These are the Link Register (LR), the Count Register (CTR) and the Condition Register (CR). The BPU saves the return pointer for subroutine calls in the LR for certain types of branch instructions. This register also contains the branch target address for the Branch Conditional to Link Register instruction. The CTR has the branch target address for the Branch Conditional to Count Register instruction. This allows for an independent execution from integer and floating -point instructions executions.

The IU executes all integer instructions. These instructions are executed one at a time. This unit resorts to the arithmetic logic unit and the General Purpose Registers (GPRs) and the Fixed Point Exception register (XER). The GPRs are used to hold the integer operands. The XER register is used for indicating conditions such as carries and overflows for integer operations.

The FPU contains a single-precision multiply-add array and the Floating-point Status and Control Register (FPSCR). The Floating Point Registers (FPRs) are used for the floating-point operations.

The LSU executes all load and store instructions and provides the data transfer interface between the GRPs, FPRs and the memory subsystem. The LSU calculates effective addresses (which will eventually be transformed to physical addresses), performs data alignment, and provides sequencing for load/store string and multiple instructions.

The SRU executes condition register logical operations, move to/from Special Purpose Registers (SPRGs) instructions and executes integer add/compare instructions.

The MMUs support up to 4 Petabytes of virtual memory and 4 Gigabytes of physical memory for instruction and data. As mentioned above the LSU calculates effective addresses which are used for data loads and stores among other things. After an effective address is generated the appropriate Memory Management Unit (MMU) translates it into physical address bits. In instruction fetches, the Instruction MMU looks for the address in the Instruction Translation Lookaside Buffer (TLB) and the Instruction Block Address Translation (IBAT). If an address can be found on both (hits), the instruction BAT's is used. When data accesses occur the MMU looks for the address in Data TLB and Data BAT. As in the instruction fetches the Data BAT is chosen in case it hits.

A.1.2 Architecture and Registers

The PowerPC architecture consists of three different layers. This approach permits for code compatibility across different implementations while at the same time allows for implementations of complexity for price/performances tradeoffs [IBM99][IBM99_A].

These three layers are:

- **User Instruction Set Architecture (UISA)** – This level defines the set of instructions and registers common to all PowerPC implementations. These include user-level registers, data types, floating-point exception model, memory models for a uniprocessor environment, and programming model for a uniprocessor environment.
- **Virtual Environment Architecture (VEA)** – This layer describes the memory model for a multiprocessor environment, defines cache control instructions, atomic operations and user-level time support.
- **Operating Environment Architecture (OES)** – This level defines privileged operations typically required by operating systems. It defines the memory management model, supervisor-level registers, synchronization requirements and exception model.

A.1 PowerPC 603e and PowerPC 750 Common Registers

The PowerPC 603e/750 registers consist of:

- 32 General Purpose Registers;
- 32 Floating Point Registers;
- Miscellaneous registers;
- Special Purpose Registers.

The PowerPC processors have two levels of privilege, supervisor mode and user mode. The former is usually used by the operating system whereas the latter is commonly used by the application software. By having privileged access to the processor's many resources, the operating system can control the applications' environment by providing virtual memory and protecting the operating system and critical machine resources. The user mode is a way of protection that limits application code from modifying sensitive resources. [IBM99] [IBM99_A] Examples of these resources are caches, memory management system and timers. The following table describes the GPRs, FPRs and the miscellaneous registers that can be found.

Registers	Description
General Purpose Registers (GPRs) – User-Level	These registers are 32 bits wide and are the source and destination of all fixed-point operations, load and store operations.
Floating Point Registers (FPRs) – User-Level	These registers are 64 bit wide. They serve as the data source and destination for floating-point instructions. These registers can hold single-precision or double-precision floating point values.
Condition Register (CR) – User-Level	This register is grouped into eight fields, each field containing 4 bits that indicate the result of an instruction’s operation. These operations can be move, compare, arithmetic and logical instructions. This register also provides a mechanism for testing and branching.

Floating-Point Status and Control Register (FPSCR) – User-Level	This register contains all exception signal bits, exception summary bits, exception enable bits and rounding control bits necessary for compliance with the IEEE 754 standard.
Machine State Register (MSR) – Supervisor Level	This register represents the state of the processor. Its contents are saved when an exception is taken and restored when exception handling completes. It also contains the settings for memory translation, cache settings, user/privileged state and floating point availability.
Segment Registers (SRs) - Supervisor-Level	In order to manage memory, there are sixteen 32 bit registers. The SRs are implemented in two ways: a main array for data accesses and a shadow array for instruction accesses.

Table A 1 - GPRs, FPRs and miscellaneous registers

Special Purpose Registers (SPRs) serve several objectives, such as giving controls, indicate status and performing special operations. As any other registers, the ability to access them depends on the access privilege of the program. Access to these registers can be explicit or implicit. This means that one can use specific instructions that move to/from a SPR or an instruction that can use a SPR as a result of a secondary operation [IBM99][IBM99_A][IBM00].

The following table describes the SPRs that can be found in the PowerPC603e/750 microprocessor.

Registers	Description
Link Register (LR) - User-Level	This register is 32 bits wide and it contains the address to return to after branch or link instructions.
Counter Register (CTR) - User-Level	The CTR is a 32 bits wide register. It contains a loop counter that is decremented and tested automatically as a result of branch operations (branch-and-count).
Fixed Point Exception Register (XER) - User-Level	This register is 32 bits wide. It contains overflow information from fixed point arithmetic operations as well as the carry input to arithmetic operations and a field

	specifying the number of bytes to be transferred by a Load String Word Indexed or Store String Word Indexed.
DSISR - Supervisor-Level	This register defines the cause of data access and alignment exceptions.
Block Allocation Table (BAT) - Supervisor-Level	There are 16 of these registers, which operate in pairs. There are four pairs Data BATs and other four pairs of Instruction BATs. These registers are used by the memory management unit to configure blocks of memory – translating logical (effective) addresses to physical (real) addresses.
Data Address Register (DAR) - Supervisor-Level	This register holds the address of an access after alignment or DIS exception.
Decrementer Register (DEC) - Supervisor-Level	This register is a 32 bit decrementing counter that provides a mechanism for causing a decrementer exception after a programmable delay.
SDR1 - Supervisor-Level	This register specifies the page table format used in virtual-to-physical page address translation.
Save and Restore 0 and Save and Restore 1 (SRR0 and SRR1) - Supervisor-Level	When an interrupt occurs, the SRR0 is loaded with the address of where processing should resume after the exception as well as the address of the instruction that caused the exception. The SRR1 is used to save machine status on exceptions. When a Return from Interrupt instruction is executed the saved values in these registers are restored.
SPRG0-SPRG3 - Supervisor-Level	These registers are provided for operating system use.
External Access Register (EAR) - Supervisor-Level	This register controls access to the external control facility through the External Control In Word Indexed and External Control Out Word Indexed instructions.
Time Base (TB) - Supervisor-Level (writing) / User-Level (reading)	This register is a 64 bit register that maintains the time of the day and operates interval timers. It consists of two 32 bit fields. These

	are the time base upper and the time base lower.
Processor Version Register (PVR) - Supervisor-Level (read-only)	This register contains the information that allows the identification of the processor. It can be very useful in case an application is programmed to act differently, depending on the type of processor.

Table A 2 - Special Purpose Registers

A.2 PowerPC 750 and PowerPC 603e comparison

In this section it will be presented main differences related to features, between the two microprocessors. Although most of the points addressed here are linked, directly or indirectly, to registers these differences should not affect the way one can program the processor. However, these features affect performance and the way instructions are executed.

PowerPC 603e	PowerPC 750
One integer unit	Two integer units. One cannot execute multiply and divide instructions.
As many as five instructions in execution per clock	As many as six instructions in execution per clock (including two integer instructions – see previous comparison)
BPU with static branch prediction	BPU with static and dynamic branch prediction
-	Use of renamed buffers (6 GPR and 6 FPR)
LRU as the used cache algorithm	PLRU as the used cache algorithm
-	Existence of L2 cache interface

Table A 3 - Features related differences between PowerPC 603e and PowerPC 750

All the features shown in the table are performance related. The PowerPC 750 can calculate more integer instructions due to its two IUs, therefore it can execute more instructions per clock cycle (if instructions with integers are being executed). The branch prediction is optimized in the PowerPC 750 by using the computation history of the program (dynamic branching), and only resorting to static when dynamic branch prediction method does not have the necessary information available. The use of renamed buffers allows the PowerPC 750 to limit stalling in integer and floating-point instructions. The used replacement algorithm in the

PowerPC 750 manages the cache in a more efficient. The existence of L2 cache allows for a lower latency when accessing the cache [IBM99].

A.2 PowerPC 750 and PowerPC 603e instruction comparison

In this section comparisons are made (regarding instructions) between the two microprocessors.

A.2.1 Instructions

The instructions used by the Xception for VxWorks-PPC603e are supported by the PowerPC 750 microprocessor and perform in the exact same way. There are, however, some differences in the implemented instructions.
The instructions that are implemented only by the PowerPC 750 are:

- **Integer Arithmetic Instructions**
  - **divdx** – 64-bit implementation of Divide Double Word.
  - **divdux** - 64-bit implementation of Divide Double Word Unsigned.
  - **mulhdx** - 64-bit implementation of Multiply High Double Word.
  - **mulhdux** – (64-bit implementation**)** Multiply High Double Word Unsigned.
  - **mulldx** - (64-bit implementation) Multiply Low Double Word.
- **Integer Logical Instructions**
  - **cntlzdx** – Count Leading Zeros Double Word.
  - **extswx** – Extend Sign Word
- **Integer Rotate Instructions**
  - **rldclx** – Rotate Left Double Word then Clear Left.
  - **rldcrx** – Rotate Left Double Word then Clear Right.
  - **rldicx** - Rotate Left Double Word Immediate then Clear.
  - **rldicl**x - Rotate Left Double Word Immediate then Clear Left.
  - **rldicrx** - Rotate Left Double Word Immediate then Clear Right.
  - **rldimix** - Rotate LeftDouble Word Immediate then Mask Insert.
- **Integer Shift Instructions:**
  - **sldx** - Shift Left Double Word.
  - **sradx** - Shift Right Algebraic Double Word.
  - **sradix** - Shift Right Algebraic Double Word Immediate.
  - **srdx** - Shift Right Double Word.
- **Floating-Point Arithmetic Instructions**
  - **fresx** - Floating Reciprocal Estimate Single.
  - **frsqrtex** - Floating Reciprocal Square Root Estimate.
  - **fselx** - Floating Select**.**
- **Floating-Point Rounding and Conversion Instructions**
  - **fcfidx** - Floating Convert from Integer Double Word.
  - **fctidx** - Floating Convert to Integer Double Word.

    - **fctidzx** - Floating Convert to Integer Double Word with Round toward Zero.
- **Integer Load Instructions**
    - **ld** - Load Double Word.
    - **ldu** - Load Double Word with Update.
    - **ldux** - Load Double Word with Update Indexed.
    - **ldx** - Load Double Word Indexed.
    - **lwa** - Load Word Algebraic.
    - **lwaux** - Load Word Algebraic with Update Indexed.
    - **lwax** - Load Word Algebraic Indexed.
    - **std** - Store Double Word.
    - **stdu** - Store Double Word with Update.
    - **stdux** - Store Double Word with Update Indexed.
    - **stdx** - Store Double Word Indexed.
- **Integer and Load and Store String Instructions**
    - **lswi** - Load String Word Immediate.
    - **lswx** - Load String Word Indexed.
    - **stswi** - Store String Word Immediate.
    - **stswx** - Store String Word Indexed.
- **Memory Synchronization Instructions**
    - **ldarx** - Load Double Word and Reserve Indexed.
    - **stdcx** - Store Double Word Conditional Indexed.
    - **stfiwx** - Store Floating-Point as Integer Word Indexed.
- **System Linkage Instructions**
    - **rfid** - Return from Interrupt Double Word.
- **Trap Instructions**
    - **td** - Trap Double Word.
    - **tdi** - Trap Double Word Immediate.
- **Processor Control Instructions**
    - **mtmsrd** - Move to Machine State Register Double Word.
- **Segment Register Manipulation Instructions**
    - **mtsrd** - Move to Segment Register Double Word.
    - **mtsrdin** - Move to Segment Register Double Word Indirect.
- **Lookaside Buffer Management Instructions**
    - **slbia** - SLB Invalidate All.
    - **slbie** - SLB Invalidate Entry. [IBM99] [IBM_99A] [IBM00] [IBM05]

The PowerPC 603e has four instructions that the PowerPC 750 does not have. These are:

- **Lookaside Buffer Management Instructions**
- **tlbld** - Load Data TLB Entry (PowerPC603e specific);
- **tlbli** - Load Instruction TLB entry (PowerPC603e specific).
- **tlbsync** - TLB Synchronize.
- **tlbie** - Translation Lookaside Buffer Invalidate Entry. [IBM99_A]

A.3 PowerPC 750 and PowerPC 603e registers

A3.1 Registers directly related to the Xception for VxWorks-PPC603e Data Collecting

Please note that in VxWorks-PPC603e plug-in the user can select several types of groups of registers (to collect data and obtain information about the consequences of fault injection) that, although related, do not belong to that group. For instance, the Integer Unit Registers includes the CR, LR and CTR, which belong to the BPU. The groups of registers are:

- Integer Unit Registers.
- Floating-Point Registers.
- Configuration Registers.
- Memory Management Registers.
- Exception Handling Registers.

Integer Unit Registers

In this unit one can find the following registers:

- GPRs.
- CR.
- XER.
- LR.
- TB.

All these registers are common (and with the same layout) to both microprocessors. The only difference is that the TB register (for reading) is in the VEA layer in the PowerPC 750 and on the PowerPC603e it is in the UISA layer. For writing, the TB register is in the OEA layer on both microprocessors [IBM99][IBM99_A].

Floating-Point Registers

The registers in this group are:

- FPRs ;
- FPSCR.

These registers can be found in the PowerPC 750 and belong to same layer (UISA) [IBM99][IBM99_A].

Configuration Registers

There are three registers in this group:

- Hardware Implementation Registers (HID);
- Machine State Register (MSR),
- Processor Version Register (PVR).

The PVR has the same functions and is in same layer. The HID are specific for each hardware implementation, so its usage and format can be different. The MSR has some differences as well. Since these registers are read by the Xception for VxWorks-PPC603e, one should be aware of its differences, so that the description of the saved information is correct. The following table shows the differences on each bit of the HID0 (HID1 is the same on both micro processors [IBM99][IBM99_A]. The symbol (-) means there is no difference.

bit	PowerPC 603e	PowerPC 750
0	-	-
1	Reserved	DBP – Enable/disable 60x bus address and data parity checking
2	-	-
3	-	-
4	-	-
5	EICE – Enables in-circuit emulator outputs for pipeline tracking	Not used
6	-	-
7	-	-
8	-	-
9	-	-
10	-	-
11	-	-
12	-	-
13	-	-
14	-	-
15	Reserved	MHR – Not hard reset. It allows software to distinguish a hard reset from a soft reset
16	-	-
17	-	-
18	-	-
19	-	-
20	-	-
21	-	-
22	Reserved	SPD – Speculative cache disable
23	Reserved	Enable M bit on bus for instruction fetches
24	Enable M bit on bus for instruction fetches	SGE – Store gathering enable
25	Reserved	DCFA – Data cache flush assist
26	Reserved	BTIC – Branch Target Instruction Cache enable
27	FBIOB – Force Branch Indirect On Bus	Not used
28	-	-
29	Reserved	BHT – Branch History Table
30	Reserved	Not used
31	-	-

Table A 4 - HID0 differences between PowerPC 603e and PowerPC 750

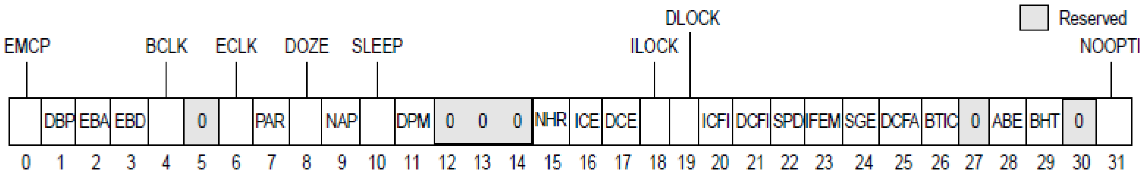


Figure A 1 – PowerPC750 HID0

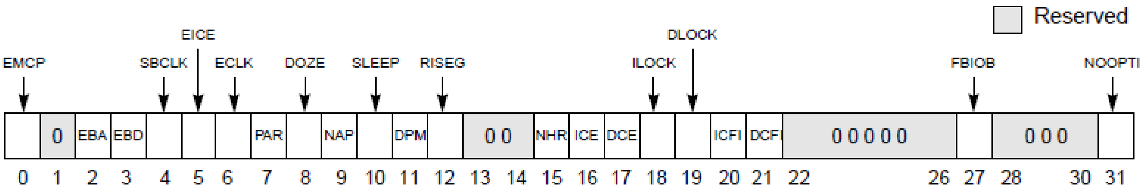


Figure A 2 – PowerPC603e HID0

The following table describes both versions of HID0 in detail.

Register	Function/Description
EMCP – PPC750, PPC603e	Enable MCP. The primary purpose of this bit is to mask out further machine check exceptions caused by assertion of MCP. 0 Masks MCP. 1 Asserting MCP causes checkstop if MSR[ME] = 0 or a machine check exception if ME = 1. [IBM99][IBM99_A]
DBP – PPC750	Enable/disable 60x bus address and data parity generation. 0 Parity generation is enabled. 1 If the system does not use address or data parity and the respective parity checking is disabled (HID0[EBA] or HID0[EBD] = 0), input receivers for those signals are disabled, require no pull-up resistors, and thus should be left unconnected. [IBM99]
EBA – PPC750, PPC603e	Enable/disable 60x bus address parity checking 0 Prevents address parity checking. 1 Allows a address parity error to cause a checkstop if MSR[ME] = 0 or a machine check exception if MSR[ME] = 1. [IBM99][IBM99_A]
EBD – PPC750, PPC603e	Enable 60x bus data parity checking 0 Parity checking is disabled.

	1 Allows a data parity error to cause a checkstop if MSR[ME] = 0 or a machine check exception if MSR[ME] = 1. [IBM99][IBM99_A]
BCLK – PPC750, SBCLK – PPC603e	CLK_OUT output enable and clock type selection. [IBM99][IBM99_A]
EICE – PPC603e	Enables in-circuit emulator outputs for pipeline tracking. [IBM99_A]
ECLK – PPC750, PPC603e	CLK_OUT output enable and clock type selection. Used in conjunction with HID0[BCLK] and the HRESET signal to configure CLK_OUT. [IBM99][IBM99_A]
PAR – PPC750, PPC603e	Disable precharge of ARTRY (Address Retry). 0 Precharge of ARTRY enabled 1 Alters bus protocol slightly by preventing the processor from driving ARTRY to high (negated) state. [IBM99][IBM99_A]
DOZE – PPC750, PPC603e	Doze mode enable. Operates in conjunction with MSR[POW]. 0 Doze mode disabled. 1 Doze mode enabled. [IBM99][IBM99_A]
NAP – PPC750, PPC603e	Nap mode enable. Operates in conjunction with MSR[POW]. 0 Nap mode disabled. 1 Nap mode enabled. [IBM99][IBM99_A]
SLEEP – PPC750, PPC603e	Sleep mode enable. Operates in conjunction with MSR[POW]. 0 Sleep mode disabled. 1 Sleep mode enabled. [IBM99][IBM99_A]
DPM – PPC750, PPC603e	Dynamic power management enable. 0 Dynamic power management is disabled. 1 Functional units enter a low-power mode automatically if the unit is idle. [IBM99][IBM99_A]
NHR – PPC750, PPC603e	Not hard reset (software-use only)—Helps software distinguish a hard reset from a soft reset. 0 A hard reset occurred if software had previously set this bit. 1 A hard reset has not occurred[IBM99][IBM99_A]
ICE – PPC750, PPC603e	Instruction cache enable

	0 The instruction cache is neither accessed nor updated. 1 The instruction cache is enabled. [IBM99][IBM99_A]
DCE – PPC750, PPC603e	Data cache enable 0 The data cache is neither accessed nor updated. 1 The data cache is enabled. [IBM99][IBM99_A]
ILOCK – PPC750, PPC603e	Instruction cache lock 0 Normal operation 1 Instruction cache is locked. [IBM99][IBM99_A]
DLOCK – PPC750, PPC603e	Data cache lock. 0 Normal operation 1 Data cache is locked[IBM99][IBM99_A]
ICFI – PPC750, PPC603e	Instruction cache flash invalidate 0 The instruction cache is not invalidated. 1 An invalidate operation is issued that marks the state of each instruction cache block as invalid without writing back modified cache blocks to memory. Cache access is blocked during this time. [IBM99][IBM99_A]
DCFI – PPC750, PPC603e	Data cache flash invalidate 0 The data cache is not invalidated. 1 An invalidate operation is issued that marks the state of each data cache block as invalid without writing back modified cache blocks to memory. Cache access is blocked during this time. [IBM99][IBM99_A]
SPD – PPC750	Speculative cache access disable 0 Speculative bus accesses to nonguarded space from both the instruction and data caches is enabled 1 Speculative bus accesses to nonguarded space in both caches is disabled. [IBM99]
IFEM –PPC750	Enable M bit on bus for instruction fetches. 0 M bit disabled. Instruction fetches are treated as nonglobal on the bus. 1 Instruction fetches reflect the M bit from the WIM settings. [IBM99]
SGE –PPC750	Store gathering enable 0 Store gathering is disabled

	1 Integer store gathering is performed for write-through to nonguarded space or for cache-inhibited stores to nonguarded space for 4-byte, word-aligned stores. [IBM99]
DCFA –PPC750	Data cache flush assist. (Force data cache to ignore invalid sets on miss replacement selection.) 0 The data cache flush assist facility is disabled 1 The miss replacement algorithm ignores invalid entries and follows the replacement sequence defined by the PLRU bits. [IBM99]
BTIC –PPC750	Branch Target Instruction Cache enable—used to enable use of the 64-entry branch instruction cache. 0 The BTIC is disabled, the contents are invalidated, and the BTIC behaves as if it was empty. New entries cannot be added until the BTIC is enabled. 1 The BTIC is enabled, and new entries can be added.
FBIOB – PPC603e	Force branch indirect on bus. 0 Register indirect branch targets are fetched normally. 1 Forces register indirect branch targets to be fetched externally. [IBM99_A]
ABE – PPC750	Address broadcast enable—controls whether certain address-only operations are broadcast on the 60x bus. 0 Address-only operations affect only local L1 and L2 caches and are not broadcast. 1 Address-only operations are broadcast on the 60x bus. [IBM99]
BHT – PPC750	Branch history table enable 0 BHT disabled. 1 Allows the use of the 512-entry branch history table (BHT). [IBM99]
NOOPTI – PPC750, PPC603e	No-op the data cache touch instructions. 0 The **dcbt** and **dcbtst** instructions are enabled. 1 The **dcbt** and **dcbtst** instructions are no-oped globally. [IBM99] [IBM99_A]

Table A 5 - PowerPC 603e/750 HID0 bit description

The next table shows the differences of the MSR between both microprocessors.

bit	PowerPC 603e	PowerPC 750
0	See bit description (Table A 7)	See bit description (Table A 7)
1	-	-
2	-	-
3	-	-
4	-	-
5	-	-
6	-	-
7	-	-
8	-	-
9	-	-
10	-	-
11	-	-
12	-	-
13	POW – Power Management Enable (603e Specific)	
14	TGPR – Temporary GPR remapping	Reserved
15	-	-
16	External Interrupt Enable	External Interrupt Enable
17	-	-
18	-	-
19	-	-
20	See bit description (Table 6)	See bit description (Table 6)
21	-	-
22	-	-
23	-	-
24	-	-
25	-	-
26	-	-
27	-	-
28	-	-
29	Reserved	OM – Performance monitor marked mode
30	-	-
31	-	-

Table A 6 - MSR differences between PowerPC 603e and PowerPC 750

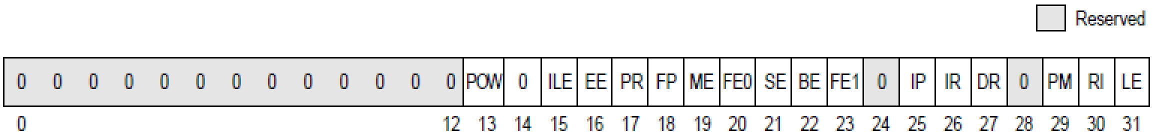


Figure A 3 - PowerPC 750 MSR

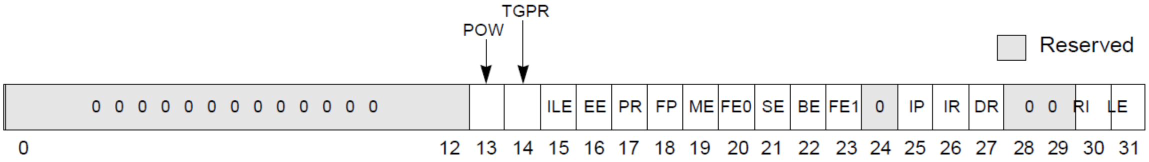


Figure A 4 – PowerPC 603e MSR

The following table describes both versions of the MSR in detail.

Register	Function/Description
POW – PPC750, PPC603e	This bit's use is to manage power. However it is implementation dependent. PPC603: 0 Disables programmable power modes (normal operation mode). 1 Enables programmable power modes (nap, doze, or sleep mode). [IBM99] PPC750: Power management enable 0 Power management disabled (normal operation mode). 1 Power management enabled (reduced power mode).[IBM99_A]
TGPR – PPC603	Temporary GPR remapping. 0 Normal operation. 1 TGPR mode. GPR0–GPR3 are remapped to TGPR0–TGPR3 for use by TLB miss routines. The contents of GPR0–GPR3 remain unchanged while MSR[TGPR] = 1. Attempts to use GPR4–GPR31 with MSR[TGPR] = 1 yield undefined results. Temporarily replaces TGPR0–TGPR3 with GPR0–GPR3 for use by TLB miss routines. When this bit is set, all instruction accesses to GPR0–GPR3 are mapped to TGPR0–TGPR3, respectively. [IBM99_A]
ILE – PPC750, PPC603e	Exception little-endian mode. When an

<table>
<tr><td></td><td>exception occurs, this bit is copied into MSR[LE] to select the endian mode for the context established by the exception. [IBM99] [IBM99_A]</td></tr>
<tr><td>EE – PPC750, PPC603e</td><td>External interrupt enable.
0 The processor ignores external interrupts, system management interrupts, and decrementer interrupts.
1 The processor is enabled to take an external interrupt, system management interrupt, or decrementer interrupt. [IBM99] [IBM99_A]</td></tr>
<tr><td>PR – PPC750, PPC603e</td><td>Privilege level.
0 The processor can execute both user- and supervisor-level instructions.
1 The processor can only execute user-level instructions. [IBM99] [IBM99_A]</td></tr>
<tr><td>FP – PPC750, PPC603e</td><td>Floating-point available.
0 The processor prevents dispatch of floating-point instructions, including floating-point loads, stores, and moves.
1 The processor can execute floating-point instructions, and can take floating-point enabled exception type program exceptions. . [IBM99] [IBM99_A]</td></tr>
<tr><td>ME – PPC750, PPC603e</td><td>Machine check enable
0 Machine check exceptions are disabled.
1 Machine check exceptions are enabled. [IBM99] [IBM99_A]</td></tr>
<tr><td>FE0 – PPC750, PPC603e</td><td>IEEE floating-point exception mode 0. (See FE1).IBM99] [IBM99_A]
<table>
<tr><th>FE0</th><th>FE1</th><th>Mode</th></tr>
<tr><td>0</td><td>0</td><td>Floating-point exceptions disabled.</td></tr>
<tr><td>0</td><td>1</td><td>Floating-point imprecise nonrecoverable. (Not Implemented in PCC603e)</td></tr>
<tr><td>1</td><td>0</td><td>Floating-point imprecise. (Not Implemented in PCC603e) recoverable</td></tr>
<tr><td>1</td><td>1</td><td>Floating-point precise mode.</td></tr>
</table>
</td></tr>
</table>

SE – PPC750, PPC603e	Single-step trace enable. 0 The processor executes instructions normally. 1 The processor generates a single-step trace exception upon the successful execution of every instruction except **rfi**, **isync**, and **sc**. Successful execution means that the instruction caused no other exception. [IBM99] [IBM99_A]
BE – PPC750, PPC603e	Branch trace enable. 0 The processor executes branch instructions normally. 1 The processor generates a trace exception upon the successful completion of a branch instruction. [IBM99] [IBM99_A]
FE1 – PPC750, PPC603e	Floating-point exception mode 1. (see FE0) [IBM99] [IBM99_A]
IP – PPC750, PPC603e	Exception prefix. The setting of this bit specifies whether an exception vector offset is prepended with Fs or 0s. In the following description, *nnnnn* is the offset of the exception. 0 Exceptions are vectored to the physical address 0x000*n_nnnn.* 1 Exceptions are vectored to the physical address 0xFFF*n_nnnn.* [IBM99] [IBM99_A]
IR – PPC750, PPC603e	Instruction address translation. 0 Instruction address translation is disabled. 1 Instruction address translation is enabled.
DR – PPC750, PPC603e	Data address translation. 0 Data address translation is disabled. 1 Data address translation is enabled.
RI – PPC750, PPC603e	Recoverable exception (for system reset and machine check exceptions) 0 Exception is not recoverable. 1 Exception is recoverable.
LE – PPC750, PPC603e	Little-endian mode enable 0 The processor runs in big-endian mode. 1 The processor runs in little-endian mode.

Table A 7 - PowerPC603e/750 MSR bit description

Memory Management Registers

There are five kinds of registers in this group:

- Instruction BAT Registers;

- Data BAT Registers;
- Set of Software Table Search Registers, SRs and SDR1.

All these registers can be found (in the same layer) in the PowerPC 750 except for the set of Software Table Search Registers. The lower Instruction BAT registers have the 28th bit available for writing in the PowerPC 750. In the PowerPC 603e doing such action would cause bounded-undefined results. Although the SRs can be found in both microprocessors, it should be pointed out that the PowerPC 750 implements these registers as two arrays (a main array for data accesses and a shadow array for instruction accesses). This should not alter any information that is gathered, because when an instruction that updates the SRs executes, the microprocessor automatically updates both sets [IBM99] [IBM99_A].

Exception Handling Registers

There are four kinds of registers in this group

- DAR;
- SPRGs;
- DSISR;
- SRRs.

Instruction Address Breakpoint Register

This register is of great importance, because it allows the software to know if the next instruction to complete is selected for fault injection. This register functions in the same way on both microprocessors. However, in the PowerPC 750 the 31th bit can be enabled so that an IABR match is signaled if this bit matches MSR[IR] (see table A 7).

A3.2 PowerPC 603e Specific Registers

There are a set of registers that are model specific (besides the HID registers). These differences should be problematic in the porting process.

The PowerPC 603e microprocessor specific registers are:

- Data TLB Miss Address Registers and Instruction TLB Miss Registers (DMISS and IMISS);
- Data TLB Compare Registers and Instruction TLB Compare Registers (DCMP and ICMP);
- Primary and Secondary Hash Address Register (HASH1 and HASH2);
- Required Physical Address Register (RPA);

The following table describes these registers.

Registers	Description
Data TLB Miss Address Registers (DMISS) and Instruction TLB Miss Registers (IMISS) - (Supervisor-Level)	These registers are loaded automatically upon data or instruction TLB miss. They contain the effective page address of the access that caused the TLB miss exception. The contents are used by the microprocessor when

	calculating the values of HASH1 and HASH2 registers and by the **tlbld** and **tlbli** instructions when loading a new TLB entry. [IBM99_A]
Data TLB Compare Registers (DCMP) and Instruction TLB Compare Registers (ICMP) - (Supervisor-Level)	These registers contain the first 32 bits in the required Page Table Entry (PTE). The contents are built automatically from the contents of DMISS and IMISS when a TLB miss exception occurs. Each entry (PTE) read from the tables during the table search process should be compared with this value to determine if the PTE is a match. [IBM99_A]
Primary and Secondary Hash Addres Registers (HASH 1 and HASH 2) – (Supervisor Level)	These registers contain the physical address of the primary and secondary PTE groups for the access that caused the TLB miss exception. These registers are constructed from the DMISS or IMISS contents (depending on the last known miss). [IBM99_A]
Required Physical Address Register (RPA) – (Supervisor-Level)	This register is loaded during a page table search operation with the second word of the correct PTE. When the adequate instruction is executed, the RPA and DMISS or IMISS register are merged and loaded into the selected TLB entry. [IBM99_A]

Table A 8 – PowerPC 603e Specific Registers

A3.3 PowerPC 750 Specific Registers

The PowerPC 750 microprocessor specific registers are:

- User Monitor Mode Control Register 0 and 1 (UMMCR0 and UMMCR1);
- Monitor Mode Control Register 0 and 1 (MMCR0 and MMCR1);
- Performance Monitor Counter Registers (PMC1-PMC4);
- User Performance Monitor Counter Register (UPMC1 – UPMC4);
- Sampled Instruction Address Register (SIA);
- User Sampled Instruction Address Register (USIA);
- Instruction Cache Throttling Control Register (ICTC);
- Thermal Management Registers (THRM1-THRM3);
- L2 Cache Control Register (L2CR).
- Data Address Breakpoint Register (DABR).

The following table describes these registers.

Registers	Description
MMCR0 (Supervisor-Level)	This register is provided to specify events to be counted and recorded. It can be accessed in supervisor mode. [IBM99]
UMMCR0 (User-Level)	This register reflects the contents of MMCR0 and can be read by user-level software. [IBM99]
MMCR1 (Supervisor-Level)	This register functions as an event selector for PMC3 and PMC4. [IBM99]
UMMCR1 (User-Level)	This register reflects the contents of MMCR1 and can be read by user-level software. [IBM99]
PMC1-4 (Supervisor-Level)	These registers are used to monitor events and can be programmed to generate interrupt signals when they overflow. [IBM99]
UPMC1-4 (User-Level)	These registers reflect the contents of PMC1-4 and can be read by user-level software. [IBM99]
SIA (Supervisor-Level)	This register contains the effective address of an instruction execution at or around the time the processor signals the performance monitor

	interrupt condition. If the interrupt is linked to an overflow, the SIA contains the instruction that caused it. Otherwise it contains the address of the last instruction that was completed. [IBM99]
DABR (Supervisor-Level)	The DABR is a 32-bit register that provides with the data address breakpoint facility that allows the detection of accesses to a designated double word. The address comparison is done on an effective address, and it applies to data accesses only. It does not apply to instruction fetches. [IBM99]
USIA (User-Level)	This register reflects the contents of SIA and can be read by user-level software. [IBM99]
ICTC (Supervisor-Level)	To avoid the complexity and overhead of dynamic clock control one can reduce the rate of instruction fetching in order to control junction temperature. The overall junction temperature reduction comes from the dynamic power management of each functional unit when the microprocessor is idle in between instruction fetches. [IBM99]
THRM1-THRM3 (Supervisor-Level)	These registers allow for control and access of the thermal management assist unit. The THRM1-2 registers provide the ability to compare the junction temperature with two user-provided limits. Having two limits allows for different actions in reducing temperature. The THRM3 register is used to enable the thermal assist unit and to control the comparator output sample time. [IBM99]
L2CR (Supervisor-Level)	This register is responsible for configuring and operating the L2 cache. [IBM99]

Table A 9 - PowerPC750 Specific Registers