

# A LEON3 Virtual Platform with real SpaceWire interfaces for dependable space software development

Antonio da Silva  
DIATEL-EUIT Telecomunicación  
Universidad Politécnica de Madrid  
Madrid, Spain  
adasilva@diatel.upm.es

Sebastián Sánchez  
Departamento de Automática  
Universidad de Alcalá  
Alcalá de Henares, Spain  
ssp@aut.uah.es

## ABSTRACT

In space software development there are strong robustness requirements that need advanced simulation techniques and tools to analyze the system behavior in the presence of faults. Even more, those simulation tools should provide the ability to communicate embedded software under development with another real working systems using standardized interfaces. In this paper, we present the design of a virtual platform for LEON3, a 32bit SPARC CPU based system used by the European Space Agency, described at Transaction Level using SystemC. By means of virtual I/O this platform allows real SpaceWire communications with another virtual or real equipment using real SpaceWire commercial hardware. Each TLM component of the model exposes a standard TLM2.0 “transport\_dbg” interface to allow internal component inspection and modification. This way full fault injection campaigns by corrupting CPU registers or memory locations can be carried out.

## Categories and Subject Descriptors

B.8.1 [Performance and reliability]: Reliability, Testing and Fault-Tolerance.

D.2.4 [Software Engineering]: Software/Program Verification – reliability, validation.

## General Terms

Design, Reliability, Verification.

## Keywords

LEON3, Fault Injection, XML Faultset, Binary Instrumentation, Debug Transport Interface.

## 1. INTRODUCTION

In today’s systems design, rapid prototyping and evaluation of expected behavior is more important than ever. Thus it is necessary to carry out testing tasks in a very early development stage to ensure that the implemented exception mechanisms work properly and helps to evaluate the risks, revealing how the system behaves in the presence of faults. These fault tolerance requirements ask for integrated, easy to use, full simulation environments, where Instruction Set Simulators (ISS) allow

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIMUTOOLS 2011, March 21-25, Barcelona, Spain

Copyright © 2011 ICST 978-1-936968-00-8

DOI 10.4108/icst.simutools.2011.245516

software to be developed and tested with a high accuracy in a very early hardware development stage. This is essential to evaluate fault detection and recovery mechanisms implemented in the software design.

Virtual platforms are software models of complete systems that provide software engineers with development environments long time before real hardware is available. Virtual platforms enable concurrent development of SoC hardware and software, significantly shortening their integration time. For embedded software development, virtual platforms provide faster edit-compile-debug cycles through more controllability, observability and determinism in the carried out experiments.

System-level modeling languages are used to start the design process from a more abstract level and apply a top-down design methodology. The use of SystemC [1] provides an industry-standard mechanism of modeling and verifying hardware and systems using standard C++ compilers. The Transaction Level Modeling (TLM)[2] raises the abstraction level description of a system, focusing in the interchange of data between components through abstract communication channels or sockets. Due to their advantages, TLMs have been traditionally used for design space exploration, early architectural performance estimations and to allow an earlier software development start, joining the hardware and software design flow.

The remainder of the paper is organized as follows: relevant related works are briefly depicted in section 2, section 3 describes the proposed framework; first of all an approach for fault taxonomy definition based on a XML Schema is introduced, next the techniques and interfaces for fault injection are described. Section 4 describes the experimental setup used to evaluate the proposed approach. Section 5 contains the conclusions.

## 2. RELATED WORK

Through the years, several fault injection techniques for fault tolerance coverage testing in critical systems have been considered. Performing fault injection by modifying physical or hardware signals at pin level [3] can only be used at very late phases of the design cycle. With hardware description languages like VHDL, others fault injection techniques can be used, especially those based on the use of “saboteurs” and “mutants” in VHDL models [4][5][6]. “Saboteurs” are additional modules inserted into the signal path between two components and “Mutants” are new versions of a module that replace the original. Normally these models describe the systems at RTL and the runtime simulation is very slow, being impossible to perform hardware-software co-simulation.

The work presented in [7] introduces fault injection methods for SystemC-based systems descriptions. One of the main drawbacks presented in many fault injection scenarios is the time overhead introduced; in order to improve the performance of executable models in the presence of faults, some strategies are used to accelerate the SystemC simulation by parallel computing.

The work in [8] presents system-level fault injection in SystemC. The proposed framework of fault injection consists on untimed functional TLM modeling with FIFO channels.

SystemC fault models and its related injection strategies are described in [9]. The approach allows the insertion of permanent and transient faults. Using Python language, a list of public features and accessible attributes of SystemC objects is extracted. Given this list, faults are inserted modifying object's attribute values. This approach limits the injection to public C++ object attributes.

SystemC provides an executable model for system description, so mutations modules can be used to test the exception handling mechanisms. Bombieri [10] proposes a mutation model for perturbing transaction level modeling (TLM) SystemC descriptions. In particular, the main constructs provided by the SystemC TLM 2.0 library have been analyzed, and a set of mutants is proposed to perturb the primitives related to the TLM communication interfaces.

Although the notion of wrapper was classically proposed for observation/detection purposes, some works like [11] have addressed the notion of wrapper to support fault injection.

SimSoC [12] presents the design of an ARM based system. The framework integrates ISSs as SystemC modules with other platform components by means of TLM interfaces. It uses dynamic translation of the binary code to speed up execution.

RESP framework [13] is so far the most complete framework for virtual platform building. The use of Python language provides a powerful mechanism to analyze the given SystemC description and extract the set of public object attributes where faults could be injected. The main drawback of this approach is that those public attributes no necessary reflect a functional view of the component, but a programmers view. Even more, good programming practices suggest those attributes to be private and provide a public API in order to access them.

The framework presented in this work is a specific LEON3 virtual platform with fault injection capabilities. First of all, a XML fault taxonomy of the fault model is proposed. This approach makes the fault description independent of application GUI and fault injection technique. Second, for components internal state corruption, we proposed the use of TLM2.0 "transport\_dbg" call in order to access components internal state. Each component must expose their internal functional attributes, allowing framework inspection and modification. For transaction level corruption we propose the use of dynamic binary instrumentation techniques (DBI) in order to insert runtime binary wrappers in the transaction path.

The use of virtual platforms have some advantages that should be highlighted: it possible to carry out many experiments in a simple an controlled way, no special equipment is necessary, it is easily adapted to a specific tested target system and hence improves embedded software quality.

### 3. FRAMEWORK DESCRIPTION

Figure 1 shows the basic structure of the proposed framework. It is build around the following components:

- Fault taxonomy and XML Fault Injection Campaign description. The proposed Schema organizes TLM faults and formalizes them in an XML based description that could be understood by different injectors. Reproducibility is an important property that guarantees the fact that another system or third party tools can perform or reproduce a fault injection experiment done previously.
- LEON3 transaction level model. The system is built around a LEON3 ISS capable of executing SPARC instruction set. This first version runs close to 5 MIPS.
- Access to internal components functional view by means of "transport\_dbg" interface. This approach can be seen as a Reflection Interface definition. Reflection is the capability of any object to provide information about its internal state in order to inspect or even modify it.
- Runtime wrappers insertion technique. It is important to highlight that all the system architecture under test is built thinking only in its functionality. Transactions wrappers can be inserted/removed later depending on the test cases without top module source code modification. The wrappers inserted in this case will inject the TLM2.0 faults defined in the XML description. If no fault description is given a golden run is obtained. This golden run acts as a reproducible reference run of the system for a particular test case.

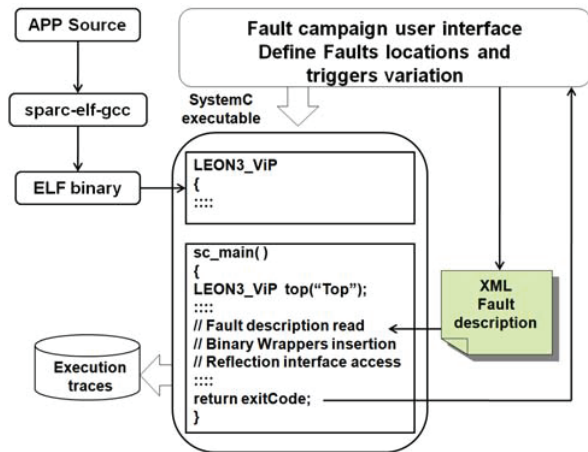


Figure 1. Proposed Framework.

#### 3.1 Basic Fault taxonomy and XML formalization

A good characterization of a fault model should allow it to be as versatile as possible, allowing a large number of combinations between the location, trigger conditions, kind of fault and duration, so that the coverage is complete. The fault model characterization is carried out by limiting where, when, for how long the fault or the corruption has to be injected and which kind of corruption has to be made.

These key attributes are taken from FARM model introduced in [14] and are formalized in a XML based description.

The work [15] describes a fault's taxonomy for binary and resources of host based systems. This approach can be used to describe memory and ISS registers corruptions by applying different kind of corruptions patterns like bitflips or bitmasks. Corruptions are those that may be caused by natural phenomena like Single Event Upsets (SEU) or Electromagnetic Compatibility (EMC) interferences and can cause bit changes in memory, CPU registers or in system buses.

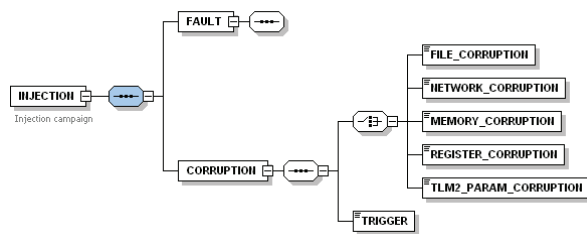
Finally, triggers are used to define when the fault must be injected and how long it takes. It describes all the conditions that should be met for a fault to become activate or deactivate. This allows defining not only permanent and transient faults but also intermittent faults.

In order to complete the approach described before, a new corruption's taxonomy for TLM2.0 transaction interface is described in Table 1. Three corruption patterns have been defined: masking, bitflipping and assigning an ad-hoc value. These corruptions patterns can be applied to the TLM2.0 parameters of blocking/non blocking calls: "b\_transport", "nb\_transport\_fw" and "nb\_transport\_bw".

**Table 1. TLM2.0 Transactions Payload Corruptions**

Where	What	When	Last
b_transport call -Command -Address -Data -Response status	Mask value BitFlip value Ad-hoc value	Always Time Trigger Signal Trigger Transact. number Logical Combo	Always Time Trigger Signal Trigger Transact. number Logical Combo
nb_transport_fw call -Command -Address -Data -Response status -Phase -TLM2_Sync	Mask value BitFlip value Ad-hoc value	Always Time Trigger Signal Trigger Transact. number Logical Combo	Always Time Trigger Signal Trigger Transact. number Logical Combo
nb_transport_bw call -Command -Address -Data -Response status -Phase -TLM2_Sync	Mask value BitFlip value Ad-hoc value	Always Time Trigger Signal Trigger Transact. number Logical Combo	Always Time Trigger Signal Trigger Transact. number Logical Combo

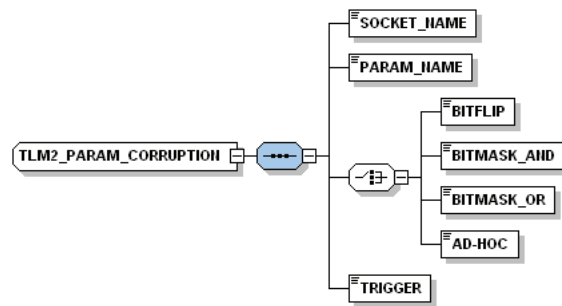
XML is a simple, well documented, straightforward data format that provides the explicitness required by machines to process data. Therefore a XML Schema has been used to define the vocabulary of the faults and the corruptions to inject and build a model of the fault domain of a system and to get a smooth integration without any conflict. From the fault and corruption taxonomy given in [15], an extension in order to support binary corruptions of TLM2 transaction parameter has been carried out.



**Figure 2. XML Schema Main tags relationship**

It has been added a new sequence of labels to the previously defined schema, see Figure 2. For the framework described in this paper, only the last three make sense:

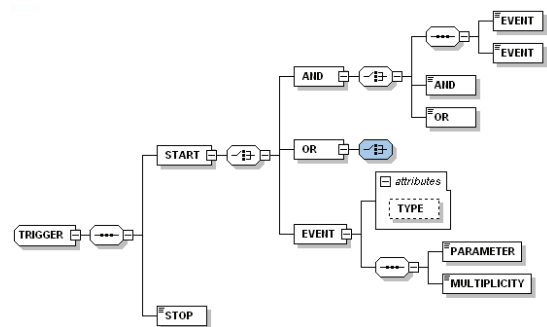
- Memory corruptions just by applying some kind of corruption pattern (bitflip, bitmask, ad-hoc new value) on a memory address location.
- Register corruption. The same as memory corruption but on CPU registers.
- TLM2 transactions. This is the new corruption sequence introduced in this work. It can be used to simulate bitflips or stuck-at faults in the buses of the model.



**Figure 3. TLM2 Parameter corruption tag structure**

Each "TLM2\_PARAM\_CORRUPTION" sequence is made of the following elements:

- SOCKET\_NAME: Hierarchical name of the TLM2.0 socket whose transactions are going to be corrupted. This name is assigned during the elaboration phase of the system.
- PARAM\_NAME: Name of the transaction parameter to corrupt as are described in OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL.
- Given a mask value on the data field of the tag several corruption patterns can be applied. In this case there are three options: BITFLIP: the active bits in the mask specifies which bits in the transaction parameter are going to be flipped, BITMASK\_AND: transaction parameter and mask are AND masked, these corruptions are useful for stuck at zero faults simulation, BITMASK\_OR transaction parameter and mask are OR masked, these corruptions are useful for stuck at one faults simulation. In the case of AD-HOC tag, the mask value overwrite the original parameter value.



**Figure 4. Trigger tag structure**

Each corruption sequence has a trigger which define when the corruption takes place and how long it lasts. A trigger consists of an event or a logical combination of events to start and an event or a logical combination of events to stop. If no trigger is provided, the corruption is active from the beginning and last forever. If there is no start event, it is assumed that the corruption starts from the beginning and if there is no stop event, it is assumed that the corruption is always active.

To configure an event it is necessary to set the type of the event (TYPE), the number of times that the event happens before it triggers the start/stop condition (MULTIPLICITY) and an extra parameter to adjust each kind of event. Trigger and event tags structure are shown in figure 4. The allowed events are shown in table 2.

Table 2. TLM2 valid events

Event types	Meaning
TIME	Time since application start in milliseconds
TLM2_TRANSACTION	Generic transaction on a given socket
TLM2_READ_TRANS	Read transaction on a given socket
TLM2_WRITE_TRANS	Write transaction on a given socket

As an example, figure 5 shows the corruption definition of the "address" parameter in a TLM2 transaction carried out through "top0.socket\_0" socket. In this case the applied corruption pattern is BITMASK\_AND and the mask value is 0xFFFFFFFF. This means that during the activation of the corruption the less significant bit of the address is set to zero. Without stop trigger that imply a permanent stuck at zero fault in the address bus. If a stop trigger is specified a transient stuck at zero fault is simulated.

```

1. <CORRUPTION>
2. <SOCKET_NAME>top0.socket_0/<SOCKET_NAME>
3. <PARAM_NAME>address/<PARAM_NAME>
4. <BITMASK_AND>0xFFFFFFFF/<BITMASK_AND>
5. <TRIGGER>
6. <START>
7. <OR>
8. <EVENT TYPE="TIME">
9. <PARAMETER>100/<PARAMETER>
10. <MULTIPLICITY>1/<MULTIPLICITY>
11. </EVENT>
12. <EVENT TYPE="TLM2_TRANSACTION">
13. <PARAMETER>module1.socket_1/<PARAMETER>
14. <MULTIPLICITY>10/<MULTIPLICITY>
15. </EVENT>
16. </OR>
17. </START>
18. </TRIGGER>
19. </CORRUPTION>

```

Figure 5. Trigger tags relationship

As for the triggers, there is a start but no stop trigger so the specified corruption last forever. The complete start trigger is a logical combination of two single events:

- A time event that fires 100 milliseconds after simulation starts.
- A transaction operation on a TLM2.0 socket given its name. The multiplicity tag establishes that the operation must take place 10 times in order to activate the event.

The final activation of the trigger is the logical OR combination of both events, so whichever takes place first activates the corruption.

### 3.2 LEON3 Transaction Level Model

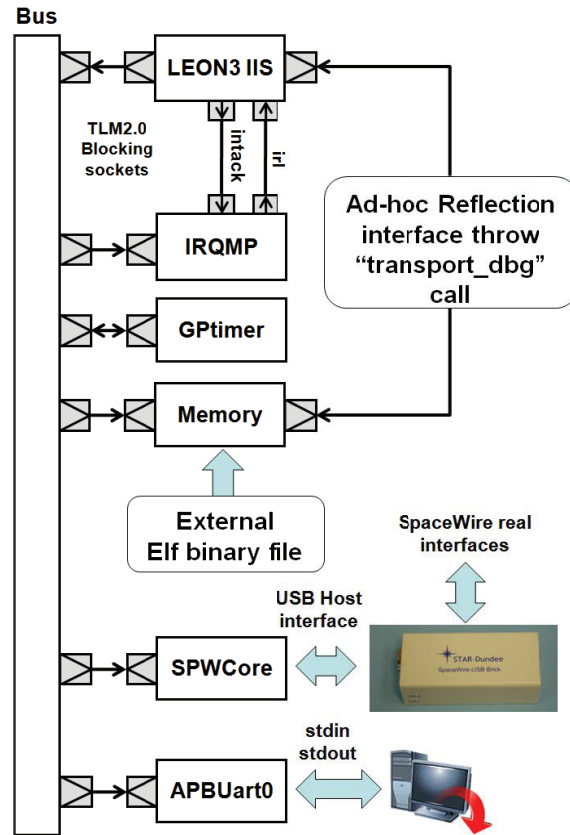


Figure 6. LEON3 basic Model.

Figure 6 show the transaction level model of the LEON3 system. The minimal building blocks are:

- LEON3 ISS: Sparc V8 untimed Instruction Set Simulator with blocking TLM2 transaction interface.
- GPTimer: Basic timer capabilities. This module is clocked by the system clock, when the prescaler underflows a timer tick is generated.
- IRQMP: Interrupt controller.
- Memory: 2Mbyte PROM beginning at address 0x00000000 and 4Mbyte RAM starting at 0x40000000. The memory contents are read from an external ordinary ELF file generated by the compiler toolchain.
- SPWCore: This interface is provided for SpaceWire communications. SpaceWire [16] is a standard for high-speed links and networks for use onboard spacecraft. By means of an external real spacewire hardware based on a Star-Dundee USB Spacewire brick [17], software running on the virtual platform can perform real spacewire communications.

- APBUart0: This interface is provided for serial communications. The UART supports data frames with 8 data bits, one optional parity bit and one stop bit. It is used by software to perform standard input/output.

Figure 7 shows LEON3 ViP running the paranoia test [18]. This test was originally written by William Kahan in the 1980’s and test several aspects of floating-point operations. The source code is distributed along with the “C” toolchain provided by Gaisler Research [19].

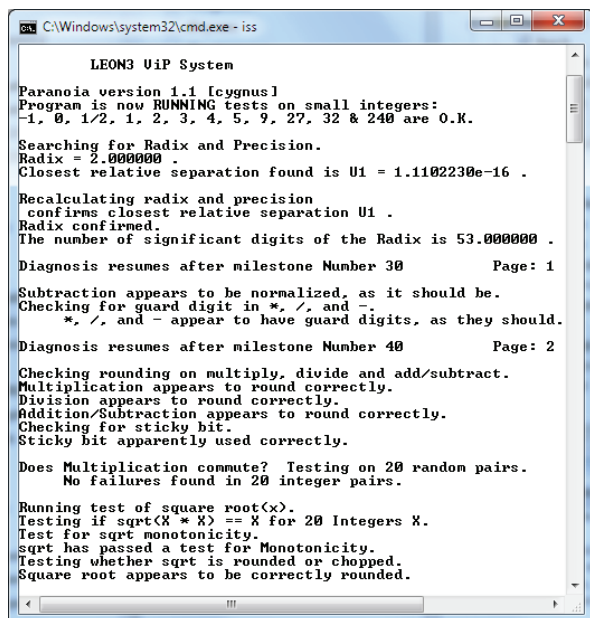


Figure 7. LEON3 ViP Running Paranoia test.

### 3.3 Reflection Interface definition

Every LEON3 ViP module with fault injection capabilities exports an ad-hoc reflection interface accessible by means of TLM2.0 “transport\_dbg” call. Through this interface the components expose a functional view of the internal architecture of the component regardless how the component has been coded. Carrying out “read/write” operations is possible to get/set the actual state of the component. For memory modules the functional view reflects the memory map. For LEON3 module, reflection interface allows accessing the architectural elements of the ISS like program counters, registers set and so on.

Table 3. Leon3 Reflection Address Map

Address	ISS Resource
0x0	G0 register
0x4	G1 register
0x8	G2 register
.....	...
0x80	PC register
0x84	NPC register
.....	...

Table 3 shows how reflection interface looks like for LEON3 module. Basically, each SPARC register is mapped as an address allowing read/write of its value through the defined interface.

### 3.4 Runtime Transaction Wrappers Insertion

For interface fault injection, transaction wrappers in TLM2.0 sockets interface can be inserted at runtime without top module modifications, depending on the specific fault set to be injected and described by the XML file. After the elaboration phase, all the elements are instantiated, the initiators sockets have been bound to targets sockets. Next and before the simulation starts, runtime wrappers are inserted using a novel virtual table hooking technique described in [20][21]. This technique can be applied after the elaboration phase is done and needs neither source code modifications nor recompilation of top level models descriptions. Once the transaction path is intercepted by means of a runtime wrapper, this can be used in several kinds of scenarios like: transaction tracking or snooping, fault injection by corrupting transaction parameter, transaction’s assertions insertion, protocol verifications, etc.

## 4. EXPERIMENTAL SETUP

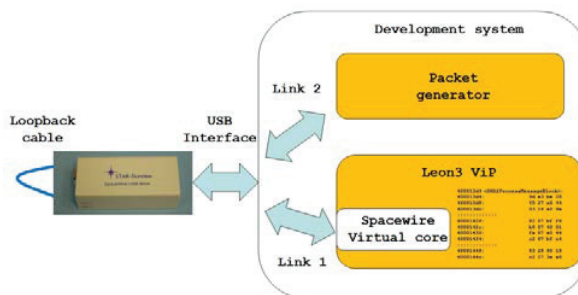


Figure 8. LEON3 ViP/Spacewire experimental setup.

In order to test the overall proposed framework, the scenario showed in figure 8 has been implemented. As target application the system runs a SHA-1 hash calculator. The SHA hash functions are a set of cryptographic hash functions designed by the National Security Agency (NSA)[22] and published by the NIST as a U.S. Federal Information Processing Standard. SHA algorithms set are intended to be employed in a wide variety of security applications, and are also commonly used to check the integrity of files. They are supposed to replace the well known MD5 hash. This kind of application is chosen because it has a high processing workload and it is easy to perform the calculation of a given input data and therefore knowing the expected results. The original source code is compiled using the “C” toolchain provided by Gaisler Research [19]. A development system runs 2 applications instances: a LEON3 ViP system and a data packet generator. Both are systems communicate across an external spacewire interface based on a commercial hardware. For each execution of the LEON3 virtual system runs as follow:

- LEON3 application under test sends a packet request across the spacewire interface.
- The Packet generator answers with a data packet containing a SHA signature at the end.
- LEON3 application under test performs the SHA hashing of the received data in order to check the data integrity. This operation is done under the fault injection campaign described in the following paragraphs.

### 4.1 Fault Model

For each memory section present in the ELF binary a complete fault injection campaign is carried out, sweeping memory address from the beginning to the end of the section. For injection time the same approach is used. From zero to end simulation time several injection points are uniform distributed along the execution time. This way the temporal and spatial influence of the faults can be explicitly stated. For each address and injection time the corresponding XML fault description is written, the virtual platform is launched and the exit code obtained, see figure 9.

```

1. For simTime=0;simTime<=END_SIM_TIME;simTime+=TIME_STEP
2. For address=BEGIN_HEAP;address<=END_HEAP;address+=4
3. Write XML fault specification
4. Launch Leon3 Virtual Platform
5. Get exit code
    
```

Figure 9. Leon3 ViP testing procedure.

Figure 9 shows the simple algorithm used in the fault campaign for HEAP section. Each memory location of the section is corrupted at several simulation times. The total number of injection times is defined by the TIME\_STEP parameter.

### 4.2 System behavior

For each fault injected in the model, the behavior of the application under test is checked. Each exit code describes one of following behaviors:

- No influence: Application runs and ends properly. The calculated hash is as expected.
- Bad hash: Application seemingly runs and ends properly but the result hash is wrong. This is the worst situation since in a real scenario there is no hint that there was a failure. Analyzing the results can help in protecting the most sensitive parts of the application.
- Memory read/write error. The faults lead the system to out of memory space access due to a bad address calculation. This kind of failures is detected and can be corrected. In the worst case a system reset is issued.
- Application Hang. The injected fault leads code to an endless loop. This situation can also be detected by watch-dog like mechanisms.
- Integer Unit (IU). This kind of failures happens only when fault affects TEXT section. In this case the original opcode is modified and the new one is an invalid opcode.

The bad hash and application hang happen not only when variables change in an unexpected manner, but also when the opcode's corruption produces a different but valid opcode. Let imagine that the executed code were the one presented in figure 10, where the memory address 0x40001428 would correspond with the assembly instruction "add %fp, -8, %g1". This instruction adds an immediate value to %fp register (frame pointer; conventionally same as %i6) and stores the result in %g1 register. Flipping just one bit, the instruction "add" becomes "sub". The final code instead of adding subtracts. Depending on the use of the operation result application can behaves in different ways. If the result is used in a loop control or memory pointer variable application could hang or perform a bad memory access. If result takes part of SHA hash calculation the result will be a bad hash.

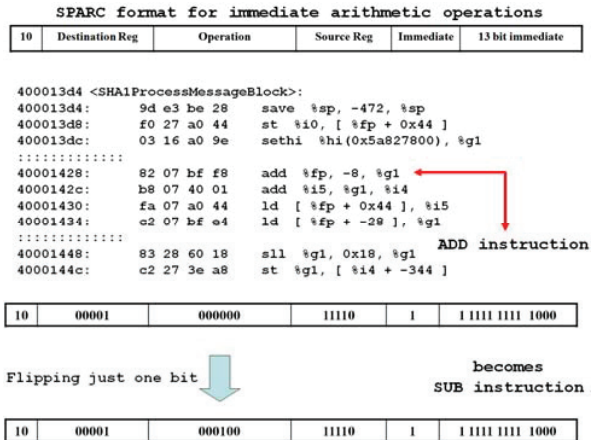


Figure 10. Sparc BitFlip

### 4.3 Fault Overall results

Table 3 describes the overall results of the 268317 experiments carried out in the fault injection campaign. Faults were injected across all different memory sections: BSS, DATA, HEAP and TEXT. The larger the memory section the greater the number of faults injected. The BSS section contains all global and static variables that are initialized to zero by default. On the other hand, DATA section contains global and static variables that are not initialized to zero. The last data section is the HEAP area which begins at the end of the BSS segment and grows to larger addresses available on the system. Finally the TEXT section contains the application code. For all the faults injected only about 17000 have some kind of effect on the program behavior. According to the results obtained the most sensitive section is TEXT.

Table 3. Overall results

Section	Injections	No Manifest	Faults	% Faults
BSS	5502	5502	0	0
DATA	15519	14481	38	0,24 %
HEAP	22260	21568	692	3,1 %
TEXT	225036	207899	16725	7,4 %
All Sections	268317	249450	17455	6,5 %

## 5. DETAILED RESULTS

The previous global results are depicted in the following lines. Individual results for each memory section are shown. No chart is given for the BSS section since all the injected faults have no influence in the application under test. This behavior could be due to BSS holds no initialized data and the corrupted values are overwritten by the program before using them. If the injection TIME\_STEP were smaller, it would be most likely to find instants when BSS's locations hold live variables and, hence, the corrupted values would be used, causing a faulty execution.

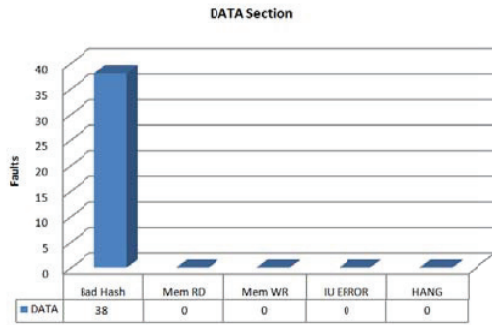


Figure 11. DATA section fault injection results.

There are only bad hash failures for faults injected in DATA section, see figure 11. This is an expected result since DATA section contains global variables and probably is used for temporary results during the final hash calculation. As was previously highlighted this is the worst behavior, so this section must be especially protected.

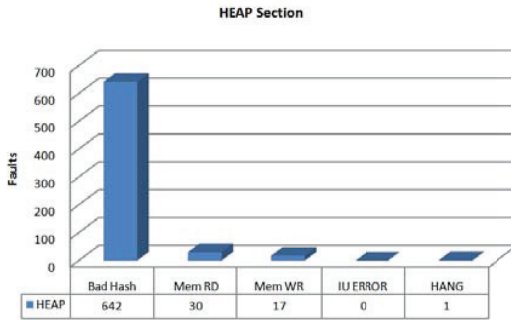


Figure 11. HEAP section fault injection results.

HEAP section, figure 11, experiments several kind of failures. Most of them are bad hash like DATA section. However there are 47 out of memory access and 1 application hang. In these cases the injected faults have effect on memory pointers and control loop variables.

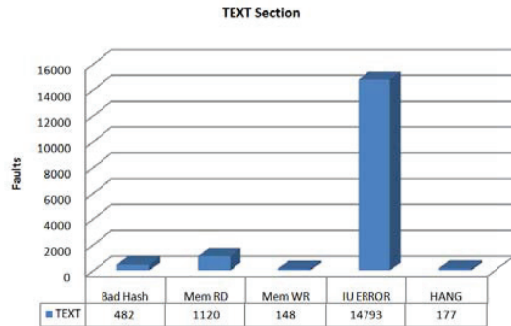


Figure 12. TEXT section fault injection results.

TEXT section results are shown in figure 12. In this case most of the failures are IU Errors. This means that opcode corruptions usually generate invalid opcodes. However there are a wide number of situations where the new opcode is a valid one and the

side effects are unpredictable. Above 1250 out of memory access and 177 application hang are the failures detected by hardware. Again the worst situations are the 482 faults that provoke no hardware errors but bad calculations.

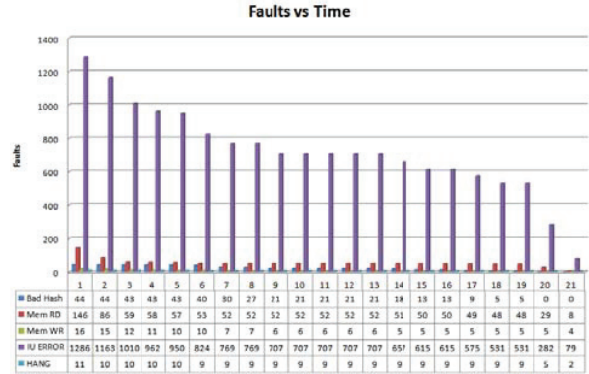


Figure 14. Failures vs time injection.

Finally figure 14 shows the distribution of the failures against the simulation time. IU failure follows a decreasing line trend from the beginning until the end of simulation time. This is an expected result since as simulation time progress most of the corrupted opcodes will never be executed. The other failures show a uniform distribution along the simulation time. Maybe the failures are a little bit higher in the middle of the simulation where the hash calculation is more intense.

As is shown in the framework description, see figure 1, for each experiment an execution log is generated. Therefore is possible to perform a postmortem analysis of the faulty execution and confront it with the golden run, in order to see where the fault begins to manifest and how it propagates on the software. Figure 15 shows an extract of execution logs and where a corrupted memory value is loaded in an ISS register.

```

Golden run
400015e0 : LD [I6 + 0x00000044], G1 [0x403FFC50+0x00000044]=0x403FFD90
400015e4 : LD [G1 + G0], G1 [0x403FFD90+0x00000000]=0x67452301
400015e8 : ST G1, [I6 + 0xFFFFFE9C] 0x67452301=[0x403FFC50+0xFFFFFE9C]

Faulty run
400015e0 : LD [I6 + 0x00000044], G1 [0x403FFC50+0x00000044]=0x403FFD90
400015e4 : LD [G1 + G0], G1 [0x403FFD90+0x00000000]=0x64452301
400015e8 : ST G1, [I6 + 0xFFFFFE9C] 0x64452301=[0x403FFC50+0xFFFFFE9C]

```

Figure 15. Fault Injection execution traces example

## 5. CONCLUSIONS

The capacity of injecting faults is essential for the verification of fault tolerance mechanisms that are foreseen in the construction of critical systems, as well as to predict the consequences of the bad systems behavior due to errors not detected in functional tests.

Efficient faultset descriptions between different fault injection tools require capturing and translating the specific vocabulary and data structures into precise and structured domain-specific descriptions. These can be achieved by the use of standard markup languages that allow giving a precise definition and meaning to fault injection terms. This approach minimizes the loss of information from one tool to another and allows efficient sharing of fault injection experiments descriptions.

This is a first modeling approach of basic fault sets in mixed TLM models descriptions and components internal state modification. It can be easily extended to other faults and corruptions scenarios. XML fault sets are easy to read and compare not only for machines but also for humans. It facilitates communication and collaboration and improves experiment reproducibility.

The runtime wrapper insertion technique used in the framework presented here shows that it is possible to insert transaction “saboteurs” in an easy way with a minimal time overhead and with a great improvement over previous approaches like interposition modules. In addition, the technique here adopted is applicable to third party software component validation, without having access to component source code. This feature greatly reduces deployment-time for real-world testing scenarios. In addition, since insertion/removal of such “saboteurs” is time-consuming and error-prone, automating this task also ensures that the testing process does not compromise the correctness of the final system.

In order to corrupt the internal state of the model’s components, the definition of well known interfaces is needed. In order to perform controlled corruptions in architectural components’ elements it is necessary to access functional views of those components.

Analysis of experiments’ results indicates that it is feasible to identify strategic locations where faults have more catastrophic consequences and hence improving the software fault tolerance.

## 6. ACKNOWLEDGMENTS

This work has been partially supported by the MICINN under the grant AYA2009-13478-C02-02.

## 7. REFERENCES

- [1] Open SystemC Initiative, <http://www.systemc.org>. SystemC.
- [2] Open SystemC Initiative, <http://www.systemc.org/downloads/standards/tlm20/>.
- [3] Benso, A., Prinetto, P., eds., “Fault Injection Techniques and Tools for VLSI reliability evaluation”, Kluwer Academic Publishing, 2003.
- [4] Jenn, E., Arlat, J., Rimén, M., Ohlsson, J. and Karlsson, J., “Fault Injection into VHDL Models: The MEFISTO Tool,” in Proc. 24th IEEE Int. Symp. On Fault-Tolerant Computing (FTCS-24), Austin, TX, USA, 1994, pp. 66-75.
- [5] Baraza, J.C., Gracia, J., Gil, D., Gil, P.J., “Improvement of Fault Injection Techniques Based on VHDL Code Modification”, Proceedings of the High-Level Design Validation and Test Workshop, 2005, pp. 693-706.
- [6] Gracia, J., Saiz, L.J., Baraza, J.C., Gil, D., Gil, P.J., “Analysis of the influence of intermittent faults in a microcontroller,” 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2008, pp.1-6
- [7] Misera, S., Vierhaus, H. T., Sieber, A., “Fault Injection Techniques and their Accelerated Simulation in SystemC”, Proceedings of the 10th Euromicro International Conference on Digital System Design, DSD 2007, pp. 587-595
- [8] Chang, K.J., Chen, Y.Y., “System-Level Fault Injection in SystemC Design Platform”, Proceedings of the 8TH Symposium on Advanced Intelligent Systems, 2007
- [9] Bolchini, C., Miele, A., Sciuto, D., “Fault Models and Injection Strategies in SystemC Specifications”, Proceedings of the 11th Euromicro International Conference on Digital Systems Design. DSD 2008, pp. 88-95.
- [10] Bombieri, N., Fummi, F., Pravadelli, G. “A Mutation Model for the SystemC TLM 2.0 Communication Interfaces”, Proceedings of the conference on Design, automation and test in Europe, 2008, pp. 396-401.
- [11] Rodríguez, M., Fabre, J.-C. and Arlat, J., “Building SWIFI Tools from Temporal Logic Specifications,” in Proc. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN-2003), San Francisco, CA, USA, 2003, pp. 95-104.
- [12] Helmester, C., Joloboff, V., “SimSoC: A SystemC TLM integrated ISS for full system simulation”, Proceedings of International Asia Pacific Conference on Computer Architecture and Systems, 2008.
- [13] Beltrame, G., Fossati, L., Sciuto, D., “ReSP: A Nonintrusive Transaction-Level Reflective MPSoC Simulation Platform for Design Space Exploration”, in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2009, Volume: 28 Issue:12, pp. 1857 – 1869.
- [14] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.-C., Martins, E., and Powell D. (1990) “Fault Injection for Dependability Validation: A Methodology and some Applications”, IEEE Transactions on Software Engineering, Vol. 16, No. 2, pp. 166-182.
- [15] Da Silva, A., Martínez J.F., Lopez, L., García A.B., Hernández, V., “XML Schema Based Faultset Definition to Improve Fault Injection Tools Interoperability”, Proceedings of International Conference on Dependability of Computer Systems. DepCos-RELCOMEX 2008, pp. 39-46.
- [16] <http://spacewire.esa.int> (2010)
- [17] <http://www.star-dundee.com> (2010)
- [18] Karpinski, R., “Paranoia: A floating-point benchmark”, Byte Magazine 10, 1985, 2 (Feb.), 223–235.
- [19] <http://www.gailer.com> (2010)
- [20] Da Silva, A., Sánchez, S., “On the use of Dynamic Binary Instrumentation to perform Faults Injection in Transaction Level Models”, Proceedings of International Conference on Dependability of Computer Systems. DepCos-RELCOMEX 2009, pp. 237-244.
- [21] Da Silva, A., Sánchez, S., “Transactions Sequence Tracking by means of Dynamic Binary Instrumentation of TLM Models”, Proceedings of the 12th Euromicro International Conference on Digital Systems Design. DSD 2009, pp. 723-728.
- [22] <http://tools.ietf.org/html/rfc1374>