Runtime Fault-Injection Tool for Executable SystemC Models

Bogdan-Andrei Tabacaru*†, Moomen Chaari*†, Wolfgang Ecker*†, and Thomas Kruse*
*Infineon Technologies AG - 85579 Neubiberg, Germany
†Technische Universität München
Firstname.Lastname@infineon.com

***Abstract*—SystemC offers valuable design, verification, and simulation advantages on different abstraction levels spanning from register transfer level (RTL) models to transaction-level modeling (TLM) including various TLM styles. The SystemC reference simulator offers limited runtime model-manipulation features, which mainly require hand-coding of fault-injection measures while developing modules and processes. This paper presents SCFIT (SystemC Fault-Injection Tool) which facilitates fault injection into executable SystemC models. SCFIT employs the GNU Debugger (GDB) controlled by Python scripts, which runs the already compiled SystemC code and allows the user to write user-friendly faulty test-case scenarios and models. The faults are injected non-intrusively at runtime through breakpoints to methods and watchpoints to member variables. SCFIT's main use is the verification of reliability related aspects and design weaknesses in safety-critical systems-on-chip (SOCs) early in the development process.**

Keywords—fault injection, system-level verification, system-on-chip, SystemC

I. Introduction

In a world in which failures in complex systems-on-chip (SoCs) may lead to loss of human life, the demand to create dependable SoCs has considerably increased in the last years. A system's dependability is characterized by a series of attributes such as a system's availability, reliability, and safety [1], [2].

The ability to execute a thorough dependability analysis on new systems has become more and more difficult because of the industry's growing demand for newer, smaller, and more complex SoCs. Newley emerged technologies (e.g., 22 nm, 14 nm) offer smaller devices and thus a larger density for the same SoC area; therefore, components (e.g., transistors, gates, registers) become increasingly more sensitive to radiation (e.g., ionic bombardment, alpha particles), aging conditions, "cross-talk" on wires, and other physical strains. This contrasts with the industry's and market's growing interest in offering increasingly safer and more reliable products. Because of this and because of costumer demand, new safety standards (e.g., ISO 26262 for automotive applications) were created. These standards oblige chip manufacturers to incorporate better and more complex error detection and correction algorithms in their systems.

Classic functional and formal verification techniques which are extremely useful to find bugs inside SoCs still under development are not sufficient to obtain a successful dependability analysis. Therefore, fault-injection techniques and error simulators have been developed to simulate the SoCs behavior under adverse conditions (e.g., temperature fluctuations, high levels of radiation) and to verify that the SoC can recover under such circumstances.

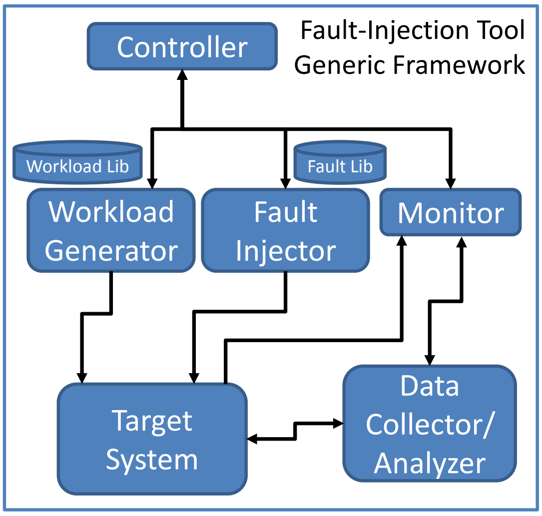
This paper proposes a new mechanism called the SystemC Fault-Injection Tool (SCFIT), which implements fault-injection capabilities in SystemC models while using the OSCI SystemC simulator. SCFIT uses the GNU Debugger (GDB) to read information from the SystemC simulator and manipulate the SystemC ports, C++ variables, and TLM payloads from the simulated models. The fault-injection scenarios are developed independently of the models and are injected at runtime. SCFIT also offers the benefit of injecting faults without having to do source code modifications.

The paper is further structured as follows: Section II describes the basics of fault injection. Section III presents a list of state-of-the-art tools for fault injection. Section IV contains the proposed tool's architecture. Section V discusses the performance results of a case study in which a SystemC testbench is simulated with and without fault injection. Section VI presents a series of optimizations which can be added to the proposed tool. Finally, section VII contains the paper's conclusions.

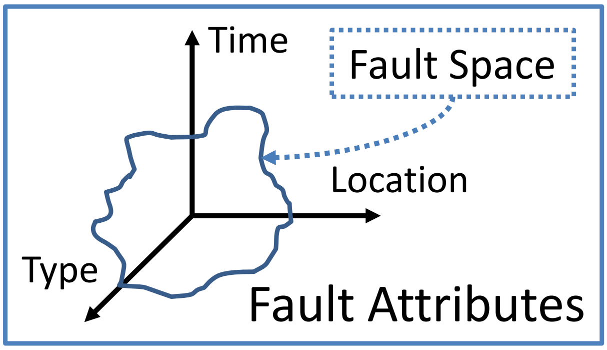
II. Fault-Injection Techniques

Fault injection is used as a dependability validation technique for hardware or software applications [1]. It is realized by implementing (or even automatically generating) testbenches by which the target system is stimulated in such a way that a faulty behavior is observed. The stimulation process is realized by an injector, whereas the observations are realized by a monitor (Fig. 1a) [2].

The injector is a configurable component similar to a regular stimulus generator and can be configured to drive new values onto variables in a target system. As opposed to a regular stimulus generator which is



(a) Generic Tool Framework



(b) Fault Space

Figure 1: Fault-Injection Basics

connected directly to the target system's inputs, the injector is also connected to internal variables. A generic injector takes as configuration arguments the attributes which make up the target system's fault space (Fig. 1b).

The fault space [1] is made up of:

- Location: the variable inside the target system where the fault should be injected.
- Time: the simulation time value or condition assertion that activates the fault injection.
- Type: the type of fault (e.g., stuck-at-0/1, single event upset, single event transient, stuck-open, bridging) which should be injected.

The monitor collects information from the target system, it analyzes it and converts it into relevant information such as coverage reports, fault-detection reports, fault-propagation reports.

The workload generator is intended to generate testbenches that stimulate the target system. The generated stimuli are used to simulate the occurrence of a specific fault or group of faults.

The controller is used to close the loop between fault injection and monitoring. The controller iterates through the generated workloads and runs multiple simulations to ensure that all relevant faults have been injected and that the target system's behavior has been monitored accordingly.

Three basic approaches to implementing fault injection are presented in [1–4]:

- Hardware-implemented fault-injection (HWIFI): utilizes specially created hardware, which allows the implementation of faults in the respective model (e.g., scan-chain, pin-level). It can be intrusive/with contact (e.g., scan-chain, pin-level) or non-intrusive/without contact (e.g., ionic bombardment).
- Software-implemented fault-injection (SWIFI): uses additional code and/or software tools to intrusively emulate fault injection in the model either by manipulating objects (e.g., signals, ports, payloads) in the model or by modifying the modeled code altogether (e.g., simulation platforms, code mutators).
- Simulation-implemented fault-injection: uses commands provided by the simulator or debugger to non-intrusively add faults in the model. The models are compiled only once and faults can be injected repeatedly in any sequence during a regression run by simply changing the test-case stimuli.

Some of the advantages of simulation-based fault injection are:

- Increased portability.
- Fast development.
- Reduced model complexity.

As all fault-injection implementations, the simulation-based fault-injection technique suffers from an added simulation overhead.

There are two types of fault-injection techniques with respect to software/hardware changes [1]:

- Intrusive: the original software model or hardware is modified to enable fault injection. Hardware/software components (e.g., method calls, objects, or processes) are either newly implemented or replaced to corrupt the original information.
- Non-intrusive: the original model's code or hardware remains unchanged throughout the whole simulation/test and analysis process.

III. Related Work

Fault injection has become a popular topic in the last decades and this has ultimately led to the release of an increasing number of fault-injection tools. With respect to SystemC models, many fault-injection techniques are already available.

In [3], [4] the authors discuss advantages and disadvantages of implementing "gate-level" SystemC models similar to VHDL models. It is shown that the SystemC reference simulator is relatively not appropriate for gate-level modeling and methods for simulation speedup are provided by implementation of parallel computing. This approach is not usable on TLM models.

AMLETO [5] was developed to simulate errors in `sc_bits`. It is used to parse SystemC code and generate a mutated version of that code. Through code mutation, testbenches are automatically generated and saboteurs are introduced in the original code to allow dynamic injection of permanent and/or transient faults during a simulation. Even though this is not actually an intrusive approach, it is necessary to create new files based on the original SystemC models which then need to be compiled and simulated. The added simulation overhead has been roughly measured to be about 76% of the original simulation time without fault injection. This tool does not support TLM models and only faults in the type `sc_bits` can be simulated.

[6] presents a similar example that uses a SystemC parser to mutate the original code and to introduce additional functions that enable fault injection. The tool is used to evaluate different fault models and to generate testbenches for these analyzed SystemC models. Similar to AMLETO, this approach is not suitable for TLM models.

FAUST [7] is a different example developed to inject both transient (i.e., bit-flip) and permanent (i.e., stuck-at-0/1) faults. It uses GDB and a series of bash-shell scripts to inject faults into SystemC models. Although it makes use of GDB's features and can access private and protected data members, FAUST apparently does not support fault-injection in TLM models and can only simulate the bit-flip and stuck-at-0/1 fault models.

Another example includes ReSP [8–10], a Python-based simulation platform, developed to inject faults into TLM models by manipulating either transactions between components or internal states of components. ReSP uses Python's powerful reflection and introspection features, but is not able to non-intrusively access private and protected data members.

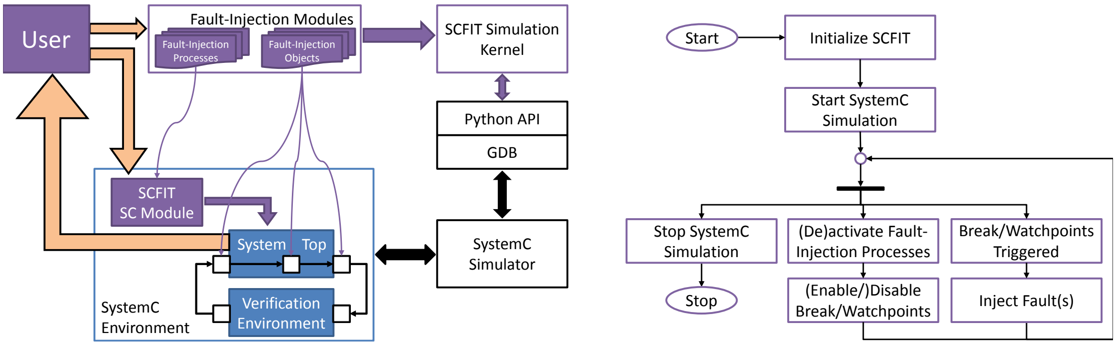
In [11] the authors present the prototype of a generic fault-injection simulator. To be able to use the simulator, new SystemC models must be programmed using the simulator's API and legacy SystemC models must be intrusively adapted to use the API. Similar to this approach, [13] proposes an error simulator which uses an extended finite state machine (EFSM) algorithm to parse and analyze SystemC models. From this analysis, test patterns and coverage metrics are non-intrusively generated. The two approaches are only applicable to clocked designs and have no documented TLM support.

Lazart [12] is a tool developed to analyze the robustness of secure smart-cards through binary-level fault injection of permanent and transient faults. It is mainly applied on C code and uses mutation to enable fault injection in the analyzed models.

The fault-injection tools presented above were successfully applied to inject faults into parts of SystemC models during simulation. When considering its implementation, FAUST is very similar to the technique presented in this paper. However, each tool has one or more disadvantages which renders them unusable on more complex SystemC and TLM models, for example mixed SystemC and TLM simulations (e.g., [3–7], [11–13]), addition of other fault models than stuck-at or bit-flips (e.g., [3–5], [7]), non-intrusive fault injection (e.g., [8–11]), etc. The proposed technique, SCFIT, is developed to bridge these limitations and allow the user to write simple fault-injection scenarios and to non-intrusively inject faults at runtime in any SystemC and TLM model, regardless of a variable's C++ access rights.

IV. SCFIT Architecture

SCFIT can be applied to non-intrusively inject faults into SystemC and TLM models. It uses breakpoints and watchpoints to create context switches between the SystemC simulation kernel and SCFIT's simulation kernel (Fig. 2a). For the sake of simplicity, in the remainder, breakpoints and watchpoints will be referred to as breakpoints.



(a) Architecture

(b) Flow Chart

Figure 2: SCFIT

SCFIT provides a user-friendly API with which the user can register C++ variable declarations, SystemC port instances, and TLM payload instances in which faults should be injected. These registered declarations and instances are called fault-injection objects (FIOs). After registering the FIOs, one or more fault-injection processes (FIPs) can be registered for each FIO.

The FIPs contain the description of the faults which need to be injected during a simulation. The implementation of FIPs is separated into a Python part, by calling the SCFIT API, and into a SystemC part, by specifying an SCFIT specific SystemC module. The FIOs and FIPs are packed into fault-injection modules (FIMs).

A FIM is a Python class that inherits SCFIT's `BaseModule` class (Listing 1). Because FIOs and FIPs can be grouped according to different criteria (e.g., according to the SystemC modules in which they are originally declared), SCFIT supports the definition of multiple FIMs. Fig. 3 presents a UML diagram of the relationship between FIOs, FIPs, and FIMs. The examples provided in Listing 1 illustrates the mapping of three FIOs (i.e., clkVar, rstVar, and storeVar) to three SystemC input ports (i.e., clk, rst, and store) and a FIP mapped to a SystemC process (i.e., io_logic) referenced to the storeVar FIO. The FIO mappings are done by manually setting the ports' C++ the hierarchical path. The FIP mapping is done by simply stating the SystemC process' name.

```
import gdb, sys , os, random
from base_module import BaseModule

class top_cpu_controller_fsm1_module(BaseModule):
  def __init__ (self):
    BaseModule.__init__(self)
    # Ports
    self.clkVar = self.addScInPort(fullPath="top.cpu.controller.fsm1.clk")
    self.rstVar = self.addScInPort(fullPath="top.cpu.controller.fsm1.rst")
    self.storeVar = self.addScInPort(fullPath="top.cpu.controller.fsm1.store")
    # Processes
    self.P1 = self.addProcess(self.storeVar, scPath="io_logic", value=0)
```

Listing 1: Fault-Injection Module Example

The actual implementation of a FIP is done in an SCFIT specific SystemC module (Listing 2). This module contains a pointer to the SystemC testbench's top module instance. For each FIP, a SystemC process shall be created. These processes are automatically executed by the SystemC kernel and process the conditions necessary to activate or deactivate the breakpoints corresponding to the FIO that references the FIP. The conditions that are processed represent either a certain simulation time value or a certain signal assertion, or a combination of both. A fault can be either injected once or until another set of conditions has been met, which disable the injection process of that fault. In the latter case, after the conditions have been met, the breakpoints are also disabled to not have unnecessary breakpoint triggers during a simulation. Finally, the FIP is also disabled.

During a normal run, GDB is configured and SCFIT is initialized. SCFIT's simulation kernel (Fig. 2a) iterates through all mapped FIOs and deduces automatically to which data types they belong (e.g., `int`, `bool`, `sc_in<int>`, `sc_out<bool>`). Depending on their data types, SCFIT creates breakpoints to their respective `read` or `write` methods, to the variables themselves, or to `(n)b_transport` TLM methods. The created breakpoints are disabled by default and only get activated and deactivated by FIPs. When the fault-injection conditions from a FIP are met, the FIP signals SCFIT which creates a context switch between its simulation kernel and the SystemC simulation kernel. During this time, SCFIT establishes the SystemC module instance

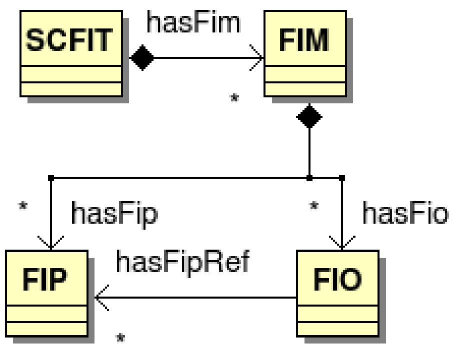


Figure 3: Relationship between FIMs, FIOs, and FIPs

and variable or data type instance in which it has to inject a fault. After fault injection, SCFIT renders control back to the SystemC simulator and the simulation continues. If the fault is injected for a longer simulation time period, then the same FIP triggers multiple context switches every time the SystemC simulator tries to override the injected value. When all faults have been injected the simulation comes to an end (Fig. 2b).

Fault injection using SCFIT is usually a straightforward process of replacing one value with another, for example, fault injection in SystemC output ports (i.e., when `write` method is called). However, injecting faults in declarations or instances which are read is more complicated because, for example, introducing a read fault in a SystemC port will affect all processes from all SystemC modules which read the `sc_signal` connected to that port. Because the fault should only affect the mapped SystemC port, SCFIT must check that the SystemC simulation kernel is reading the port from the correct SystemC module instance. This is done by calling GDB to parse and evaluate the SystemC module's `this` pointer.

Although mapping declarations and instances is also straightforward, doing so for variables declared or data-types instantiated in SystemC processes requires three steps. The first step is to create a breakpoint to the respective SystemC process. Step two requires SCFIT to figure out which SystemC module instance is handled by the SystemC simulator. This is important because faults are injected in individual SystemC module instances and not in the modules themselves. The last step is to create a breakpoint to the desired variable or data-type instance. When the SystemC simulator leaves the scope of the process in question, GDB automatically deletes the breakpoints on the declarations which means that new breakpoints must be created to inject faults into the same declarations at a later stage of the simulation. This limitation is introduced because declarations and instances inside a C++ method are invisible and inaccessible outside of the method's scope, unless they are static.

```
struct scfit : public sc_module {
  void run(); // SC_METHOD
  // Other module specific information
};

void scfit::run() {
  next_trigger(sc_time(100, SC_NS));
  if (sc_time_stamp() == sc_time(100, SC_NS)) {
    // Activate specific fault-injection processes
    next_trigger(sc_time(300, SC_NS));
  }
  if (sc_time_stamp() == sc_time(300, SC_NS)) {
    // Deactivate specific fault-injection processes
    next_trigger();
  }
```

Table I: SCFIT's Python user API

Called on	Method	Description
FIM instances	addScInPort	register a SystemC input port
FIM instances	addScOutPort	register a SystemC output port
FIM instances	addMemberVariable	register a C++ variable declared in a class
FIM instances	addMethodVariable	register a C++ variable declared in a method
FIM instances	addTlmPayload	register a TLM payload
FIO instances	addProcess	register a SystemC process and reference it to a FIO

Table II: SCFIT's Impact on Simulation Performance

SCFIT	Faults	Fault Events	CPU Time (s)	Slowdown (%)
no	0	0	3.71	0
yes	0	0	3.77	1.61
yes	1	70	4.10	10.51
yes	2	140	4.50	21.29
yes	3	160	4.61	24.25

```
}
```

Listing 2: SCFIT SystemC Module Example

SCFIT offers the possibility to create a specific fault model for each injected fault. User-defined fault models may be implemented by correctly setting up the tool's fault-injection parameters: the SystemC object's module path to be mapped, value to be injected, injection type (i.e., driven or frozen), and injection conditions. By configuring these parameters, SCFIT offers the possibility to configure the system's fault space (Fig. 1b).

Although SCFIT requires the implementation of a SystemC module inside the testbench, the presented approach remains non-intrusive because the original models subjected to fault injection are not changed. The additional SystemC module's main advantage is moving the computational requirement for validating FICs from the interpreted Python side to the compiled C++ side and thus reducing the simulation overhead.

Injected faults are either driven or frozen. Driven faults are injected once and then overridden by the SystemC simulator. Frozen faults are re-injected by SCFIT every time the SystemC simulator accesses a value (e.g., `read` or `write` method call or variable assignment) until explicitly released by a FIP. To inject faults into a multiply-instantiated SystemC module, SCFIT actively monitors SystemC's modules pool; therefore, the corresponding FIPs are activated only when the SystemC simulation kernel is executing a process from the specifically mapped module instance.

Fault injection with SCFIT is highly dependent on the amount of breakpoints created. Although one could practically inject faults into all system specific variables, ports, or payloads during the same simulation run, it is recommended that only one fault should be injected for every simulation run. This comes as a limitation provided by regular x86_64 processors which only provide four hardware breakpoints. After exceeding this number, software breakpoints may be used, however, this introduces a dramatic and undesired simulation overhead. By only injecting one fault per simulation, this limitation is successfully overcome.

V. Results

SCFIT has been successfully used to inject faults into a SystemC 32-bit CPU model developed at Infineon Technologies. SCFIT's performance has been measured using the UNIX command `time -v` (Table II).

The first line of Table II contains a simulation without SCFIT. The second line presents a simulation including SCFIT without injecting faults. The following lines represent simulations, each one containing an increasing number of injected faults. From Table II one can conclude that following statements:

- SCFIT has a static simulation overhead (i.e., simply running a simulation with SCFIT) of only 1.61%.
- SCFIT's dynamic simulation overhead is greatly dependent on the number of injected faults and on each injected fault's life time (i.e., it increases when a fault is injected for a longer period of time). This is applicable for permanent and single-event transient faults, because they are frozen during a simulation run, where as single-event upsets are only driven. Therefore, one can conclude that:
  - the longer a GDB breakpoint/watchpoint remains active, the greater the simulation overhead becomes and
  - the more GDB breakpoints/watchpoints employed, the greater the simulation overhead because the risk of surpassing the number of hardware supported breakpoints/watchpoints raises.

VI. Future Work

Although SCFIT's current features have proved successful to inject faults in SystemC models, the authors wish to further develop and improve the presented tool. Newer versions of SCFIT should include a generation tool with which users can map fault-injection objects, specify new fault models, and describe a fault's behavior during a simulation in a user-friendly way. This add-on should automatically generate the corresponding fault-injection modules, objects, and processes, as well as the SystemC module. Another important improvement for SCFIT is the implementation of an automatic fault-detection and fault-propagation analysis algorithm. This feature is meant to automatically monitor and detect injected faults in the SystemC models and recursively trace their propagation path up to the system's output. The propagation path follows the fault -> error -> failure flow.

VII. Conclusions

The presented approach allows the user to non-intrusively inject faults into TLM payloads, SystemC ports and C++ variables, regardless of the variable's access rights (i.e., public, protected, private). Legacy SystemC code can also be subjected to fault injection with SCFIT, provided that the additional SystemC module is included in the legacy SystemC testbench.

The presented tool has been successfully used to inject faults into a SystemC 32-bit CPU model developed at Infineon Technologies. SCFIT's simulation overhead was measured to be approximately 10% when injecting one fault.

Although SCFIT allows the user to inject as many faults as desired during a simulation run, due to the increasing simulation overhead it is recommended to only inject one fault per simulation. SCFIT's simulation slowdown is primarily dependent on the number of injected faults, whether faults are driven or frozen, the amount of GDB breakpoints used (i.e., the number of software breakpoints), and the number of model instances in which faults are injected.

Acknowledgment

This work is partially supported by the German Federal Ministry of Education and Research (BMBF) in the project EffektiV (contract no. 01IS13022).

References

[1] A. Benso and P. Prinetto, *Fault injection techniques and tools for embedded systems reliability evaluation*. Springer, 2004.

[2] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[3] S. Misera, H. T. Vierhaus, and A. Sieber, "Fault injection techniques and their accelerated simulation in systemc," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*. IEEE, 2007, pp. 587–595.

[4] S. A. Misera, "Simulation von fehlern in digitalen schaltungen mit systemc," Ph.D. dissertation, Universitätsbibliothek, 2007.

[5] A. Fin, F. Fummi, and G. Pravadelli, "Amleto: A multi-language environment for functional test generation," in *Test Conference, 2001. Proceedings. International*. IEEE, 2001, pp. 821–829.

[6] F. Bruschi, F. Ferrandi, M. Chiamenti, and D. Sciuto, "Error simulation based on the systemc design description language," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*. IEEE, 2002, p. 1135.

[7] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, I. Solcia, and L. Tagliaferri, "Faust: fault-injection script-based tool," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE, 2003, p. 160.

[8] F. Arlati-arlati, A. Miele, and F. Bruschi, "Resp user manual," revision 2: June 2011.

[9] G. Beltrame and L. Fossati, "Resp: a design and validation tool for data systems," in *DASIA 2008 Data Systems In Aerospace*, vol. 665, 2008, p. 22.

[10] G. Beltrame, L. Fossati, and D. Sciuto, "Resp: a nonintrusive transaction-level reflective mpsoc simulation platform for design space exploration," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 12, pp. 1857–1869, 2009.

[11] R. A. Shafik, P. Rosinger, and B. M. Al-Hashimi, "Systemc-based minimum intrusive fault injection technique with improved fault representation," in *On-Line Testing Symposium, 2008. IOLTS'08. 14th IEEE International*. IEEE, 2008, pp. 99–104.

[12] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 213–222.

[13] F. Bruschi, F. Ferrandi, and D. Sciuto, "A framework for the functional verification of systemc models," *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 667–695, 2005.