A Meta-Modeling-Based Approach for Automatic Generation of Fault-Injection Processes

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

***Abstract*—Newly emerging safety standards are driving system-on-chip manufacturers to develop better error-detection and correction mechanisms as well as verification environments for their systems. Fault injection in models is one enhancement for verification environments with which chip aging and safety-critical features (e.g., resilience against radiation sensitivity) are verified. SystemC offers the ability to design functional models on different abstraction levels (e.g., register transfer level, transaction-level modeling). Using the SystemC Fault-Injection Tool (SCFIT), faults are injected into SystemC models without modifying the original models. In this paper, we propose SCFIT Gen, an add-on for SCFIT, which automatically generates the SCFIT user-defined files and documentation about the generated fault-injection processes.**

Keywords*—*fault injection; system-level verification; system-on-chip; SystemC

I. Introduction

The question of safety has become a key point for developers of intelligent electronic applications. The development of safe products is strongly linked with the dependability analysis performed on the system under verification. This analysis is necessary for the prevention and removal of faults and errors that might appear either permanently or transiently in a system. Several safety standards (e.g., ISO 26262 for automotive applications) [1] have been created to ensure that application developers have strict quality references when designing their products. Because of the introduction of increasingly more complex error detection and correction mechanisms, the system's complexity also increases, simulation times become slower, and the verification state-space of system models becomes larger and more difficult to extensively cover.

The use of SystemC to create RTL or transaction-level models [2] for automotive applications offers a good solution to the problem of simulation speed. Nowadays, virtual prototyping has become a usual modeling approach to design and verify complex systems at an early stage in the development cycle [3]. One of SystemC's main draw-backs is the library's limited amount of features for verification. Consequently, third-party solutions are required to address these limitations.

SCFIT is a runtime-based fault-injection tool for SystemC. It was developed to enable non-intrusive fault-injection into SystemC and TLM 2.0 models during a simulation run. The faults are described using Python scripts and are injected with the GNU Debugger (GDB) through breakpoint triggers.

In this paper, the add-on SCFIT Gen is proposed for SCFIT. SCFIT Gen offers a graphical user-interface (GUI) through which SCFIT specific user-defined fault-injection processes can be defined. The proposed tool generates the corresponding SCFIT files as well as the documentation for the injected faults. This approach helps to integrate the description and documentation of the faults to be injected in a user-friendly method and to reduce the amount of user-written code necessary to enable fault injection. Updates to the fault-injection processes can be done through the GUI and re-generation is straightforward.

To the best of the authors' knowledge, SCFIT is currently the only existing tool which supports fault injection into SystemC ports, TLM payloads, and C++ variables, regardless of the data object's access rights and without changing the original model's code.

The paper is further structured as follows: Section II presents a list of tools developed to inject faults into various designs. In section III, SCFIT's architecture is introduced. Section IV describes the architecture of SCFIT Gen. Section V documents the proposed tool's results. Finally, the paper's conclusions are introduced in section VI.

II. Related Work

The existing fault-injection tools differ in many perspectives. One of these main characteristics is the tool's usability. A review on existing fault-injection tools is presented in [4]. In this paper, the authors have summarized several fault-injection tools, which are similar to SCFIT but are orthogonal to each other.

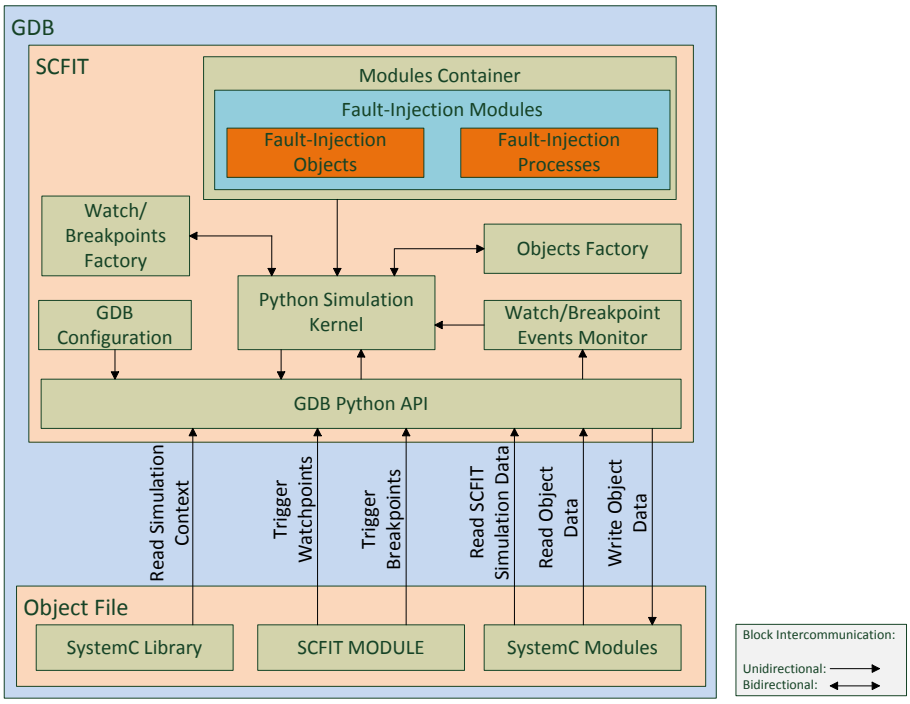


Figure 1: SCFIT Architecture

A. *ReSP*

ReSP (Reflective Simulation Platform) is a simulation platform implemented in Python [5]. The tool uses `Boost.Python` to communicate with the SystemC models. Users can inject faults only into transaction-level models. The ReSP user manual claims that the platform offers non-intrusive access to the SystemC models [6], however `Boost.Python` does not allow access to private or protected data members.

B. *FAUST*

FAUST (FAUlt-injection Script-based Tool) enables fault-injection through GDB [7]. It is used to inject constrained-random transient or permanent faults into CPU models. The tool provides a fault-injection report consisting of the messages:

- fault not effective,
- program crashed, or
- wrong output received.

C. *SystemC Debugger*

Rogin F. et al. have developed a tool to debug SystemC designs [8]. They used GDB to non-intrusively analyze a SystemC design at runtime and proposed special debugging patterns to find and fix recurring errors. The debugging tool executes a SystemC simulation up to the first error, then it exhaustively tries out all debugging patterns until the error is found. If the error is still undiscovered, the user has to manually debug the SystemC models and the C++ code.

D. *SystemC Fault-Injection Technique*

In paper [9], Shafik R. A. et al. proposed a fault-injection method for SystemC models. The authors extended the SystemC library with new signal and data types, which they used to replace the existing ones inside the SystemC models (e.g., `sc_int<N>` is replaced by `IntReg<N>`). Although this approach offers a decent simulation performance, it is intrusive because the signals' or variables' data types must be changed to allow fault injection.

The above presented tools were created to inject faults into or to debug SystemC models in different ways. ReSP can only inject faults into SystemC transaction-level models, while FAUST only in CPU models. ReSP uses a Python implementation, FAUST uses GDB. The SystemC Debugger in [8] also uses GDB but cannot inject faults. The fault-injection technique presented in [9] has a SystemC implementation, is used at compile-time,

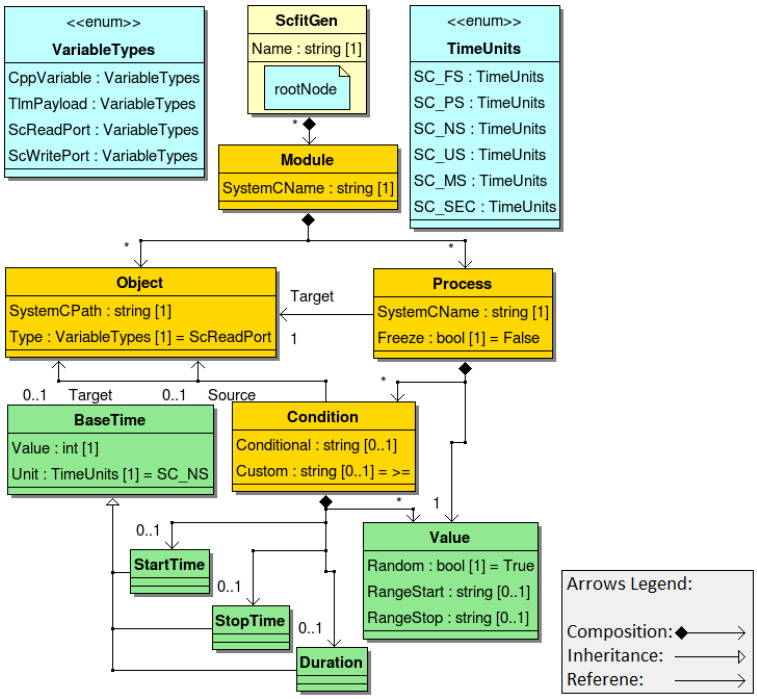


Figure 2: SCFIT Gen UML Diagram

and is intrusive. As a general rule, all tools have their own limitations depending on their application domain, supported operating system, method of implementation (e.g., software-based, simulation-based), implementation language, way in writing fault-injection scenarios, performance, intrusiveness, etc. SCFIT was created to address a part of the presented draw-backs. It is a non-intrusive tool, which can be used to inject faults in any kind of SystemC models. SCFIT's main draw-backs are the added simulation overhead by GDB (10% when injecting a single fault) and amount of user-written code necessary to describe fault-injection processes. Using SCFIT Gen, users can now describe fault-injection scenarios for SCFIT through the provided user-friendly GUI and non-intrusively inject faults in any SystemC and TLM 2.0 model during a simulation run.

III. SCFIT

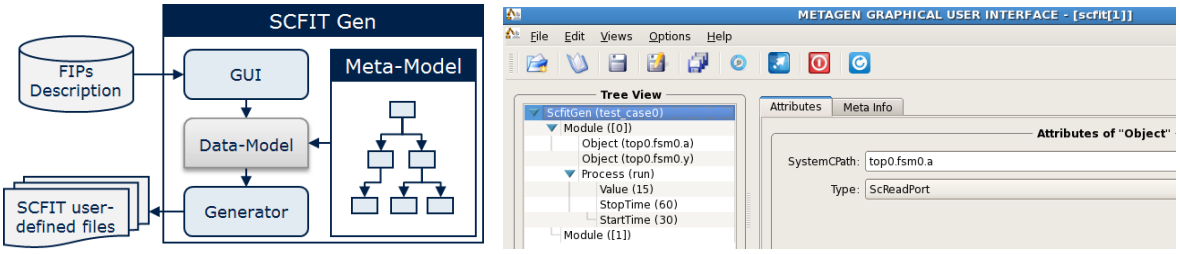
SCFIT is a runtime-based fault-injection tool created to inject faults non-intrusively into SystemC and TLM 2.0 models during a simulation run. It uses the GNU Debugger (GDB) and Python scripts to enable fault-injection and has been developed for the SystemC reference simulator. SCFIT uses GDB breakpoints and watchpoints, defined using GDB's Python API, to create context switches between the SystemC kernel and SCFIT's Python kernel (PK) (Fig. 1).

SCFIT uses GDB breakpoints and watchpoints to trigger the fault-injection mechanism into a SystemC model. The breakpoints and watchpoints are set on SystemC ports, TLM payloads, or C++ variables and work like event triggers. They trigger context-switches between the SystemC kernel and SCFIT's Python kernel (PK).

When SCFIT is initialized, the PK builds the module container (MC) and all fault-injection modules (FIMs) are registered inside the MC. FIMs contain fault-injection processes (FIPs) and instances of fault-injection objects (FIOs). FIOs are SCFIT objects that interface SystemC objects (i.e., C++ variables, SystemC ports, and TLM payloads) into which faults are injected. FIPs control FIO instances and describe how and when faults are injected into the SystemC models. The PK only activates an FIP if the FIP's fault-injection conditions (FICs) are validated. FICs are validated inside the special SCFIT SystemC module. After fault injection, the control is switched back to the SystemC kernel and the simulation continues until the next breakpoint is hit or until the simulation ends. The communication between SCFIT and GDB is done via the standard GDB Python API [10].

The initial SCFIT implementation offered a pure Python solution. This implementation introduced a considerable simulation overhead due to the expensive computation and interpretation time required by the Python code. Currently, SCFIT offers a hybrid implementation of Python and SystemC code. Compared to a simulation run without SCFIT, the hybrid implementation of SCFIT introduces a total simulation overhead of only 10% when injecting a single fault.

SCFIT's architecture is made up of three parts (Fig. 1):



(a) Flow Chart (b) GUI Example

Figure 3: SCFIT Gen

1) A set of Python scripts which functionally implement SCFIT's Python kernel (PK) and provide a user API to register information with the PK.
2) User-defined Python files which register information with the PK.
3) An SCFIT SystemC module which validates the user-defined fault-injection conditions to activate the corresponding fault-injection processes during a simulation run.

IV. SCFIT Gen

SCFIT Gen is an add-on for SCFIT and is used to automatically generate user-defined fault-injection processes (FIPs) and scenarios (FISs) for SCFIT.

The add-on provides a graphical user-interface (GUI), with which the user manually maps SystemC ports, C++ variables, and/or TLM payloads from the SystemC models to fault-injection objects (FIOs). If the manually mapped data member does not exist in the SystemC model, SCFIT issues an error message. After mapping the FIOs, the user describes within the GUI the scenarios through which faults will be injected at the simulation's runtime. SCFIT Gen will generate the Python fault-injection modules (FIMs), fault-injection object instances (FIOs), FIPs, and the SCFIT's SystemC module from the user's input data. Finally, the add-on generates a text document with information regarding the fault-injection sequences, for which the FIPs are generated. Inside the documentation file, the FIPs are ordered according to the simulation time as well as the fault-injection conditions (FICs) in which the faults need to be injected at runtime in the SystemC models.

SCFIT Gen uses a meta-model (i.e., the data-model's class-based abstract representation) to hold the user-defined information. This information is filled using the GUI. The GUI is automatically generated from the provided meta-model using Infineon's Metagen flow [11]. The meta-model's UML diagram is presented in Fig. 2. After the meta-model has been filled, its information is passed to specific mako templates that, in turn, generate the corresponding documentation, Python, and SystemC files. The add-on's generation flow is presented in fig. 3a.

A. Graphical User-Interface

The GUI offers a user-friendly mechanism to describe the FIPs that need to be injected in the SystemC models. To do so, the user must define the following items:

- **fault-injection modules**: map the SystemC modules in which faults need to be injected.
- **fault-injection objects**: map the SystemC ports, C++ variables, or TLM payloads which will be modified at runtime.
- **fault-injection processes**: map the SystemC processes or C++ methods in which the FIOs are read or written.
- **fault-injection conditions**: setup under which conditions (simulation time, signal or variable changes) a FIP should be activated.

The GUI uses an XML interchange format to fill in SCFIT Gen's meta-model. The example from Fig. 3b presents the description of a SystemC-port read-fault for the fault-injection object **top0.fsm0.a** declared in the SystemC module **fsm** when process **run** is executed. The injected value is randomly generated within the range 15 to 30 and it should be injected between 30 ns and 60 ns.

The GUI provides two methods with which fault-injection conditions can be described. First, the user could manually write the code for each SystemC if-statement inside the GUI. Alternatively, the user could use the GUI to add very simple conditions such as:

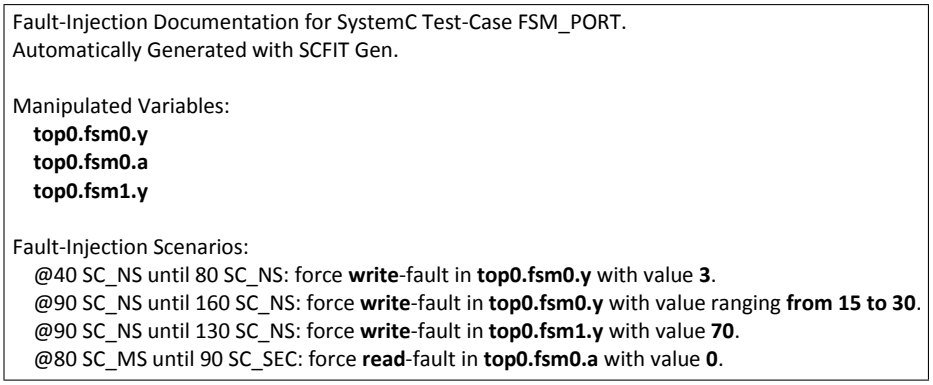


Figure 4: SCFIT Gen Documentation Example

- `[Target object] [OP] [Source object]`, where OP is any C++ binary or unary operator (e.g., Target object's value >= Source object's value).
- `[Target object] = [value]`, where value is either singular or inside a range (e.g., Target object's value = 30, Target object's value has any value between 15 and 30).

Currently, SCFIT Gen can only be used to generate FIPs and documentation for individual SystemC test-cases. This means that the tool should be independently executed for each test-case in which faults need to be injected. This results in a library of test-case dependent Python and SystemC files.

B. Generated SCFIT Files

The user-defined FIMs, FIOs, and FIPs are generated as Python files which are then used by SCFIT's Python kernel. The FICs are generated as part of SCFIT's SystemC module. The generated SystemC module must be instantiated in the SystemC test-bench. A reference to the SystemC top module must be passed to SCFIT's SystemC module during instantiation.

An example of the automatically generated documentation file is presented in CLF format in Fig. 4.

The FICs generated in SCFIT's SystemC module from Listing 1 have been described using SCFIT Gen's GUI (Fig 3b). These FICs are defined in the SystemC process `run`.

V. RESULTS

Using SCFIT Gen, the amount of user-written code is reduced by a factor of at least forty. This leads to a significant reduction in debugging time. Any necessary update for the fault-injection value or fault-injection condition is centralized within the GUI and re-generation is straightforward.

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 I).

The first line of Table I 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 I 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.

Table I: 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

VI. Conclusions

This paper introduced SCFIT Gen, an add-on for SCFIT. The proposed generation tool offers a centralized mechanism to automatically generate fault-injection processes for SCFIT and to also generate documentation for the generated FIPs and fault-injection sequences.

SCFIT Gen reduces the necessary time to setup and update fault-injection scenarios using SCFIT thanks to its powerful and user-friendly GUI. The amount of user-written code is reduced by a factor of approximately forty. The generated documentation is also directly linked to the data provided through the GUI and offers a more readable alternative to understanding the generated fault-injection processes and sequences. The documentation is also automatically updated with each new re-generation.

Currently, SCFIT Gen is being extended to generate documentation in several formats (e.g., HTML, CLF file) and to generate FIPs for multiple SystemC test-cases executed in a single run. SCFIT Gen's mechanism to define fault-injection conditions within the GUI is also being updated to allow more complex combinations.

Acknowledgment

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

```
SC_MODULE(scfit) {
    typedef enum scfit_bps_status_e {
        SCFIT_INITIALIZE = 0,
        SCFIT_FREEZE_top0_fsm0_a_p1,
        SCFIT_RELEASE_top0_fsm0_a_p1
    } scfit_bps_status_e;
    void run ();
    scfit (sc_module_name, top*);
private:
    top*                        top0;
    sc_time                     minimum_timestep;
    static scfit_bps_status_e   bps_status;
};
scfit::scfit_bps_status_e scfit::bps_status = SCFIT_INITIALIZE;
void scfit::run () {
    next_trigger(sc_time(30, SC_NS));
    if (sc_time_stamp() == sc_time(30, SC_NS)) {
        bps_status = SCFIT_FREEZE_top0_fsm0_a_p1;
        next_trigger(sc_time(60, SC_NS));
    }
    if (sc_time_stamp() == sc_time(60, SC_NS)) {
        bps_status = SCFIT_RELEASE_top0_fsm0_a_p1;
        next_trigger();
    }
}
scfit::scfit (sc_module_name mn, top* top_)
: sc_module(mn), top0(top_) {
    SC_METHOD(run);
}
```

Listing 1: SCFIT SystemC Module Example

References

[1] T. Dittel and H.-J. Aryus, *How to Survive a safety case according to ISO 26262*. Springer, 2010.

[2] F. Ghenassia *et al.*, *Transaction-level modeling with SystemC*. Springer, 2005.

[3] J. Oetjens, N. Bannow, M. Becker, O. Bringmann, A. Burger, M. Chaari, S. Chakraborty, R. Drechsler, W. Ecker, B.-A. Tabacaru *et al.*, "Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*. ACM, 2014, pp. 1–6.

[4] S. Tixeuil, W. Hoarau, and L. Silva, "An overview of existing tools for fault-injection and dependability benchmarking in grids," in *Second CoreGRID Workshop on Grid and Peer to Peer Systems Architecture*, 2006.

[5] 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.

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

[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. Rogin, E. Fehlauer, C. Haufe, and S. Ohnewald, "Debug patterns for efficient high-level systemc debugging," in *Design and Diagnostics of Electronic Circuits and Systems, 2007. DDECS'07. IEEE*. IEEE, 2007, pp. 1–6.

[9] 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.

[10] R. Stallman, R. H. Pesch, S. Shebs *et al.*, "Gdb user manual: Debugging with gdb (the gnu source-level debugger)," 2014.

[11] W. Ecker, M. Velten, L. Zafari, and A. Goyal, "The metamodeling approach to system level synthesis," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–2.