Automatic Fault Injection into SystemC Designs

Tulio Paschoalin Leao[*], Frank Sill Torres[†]
Department of Electronic Engineering (DELT),
Federal University of Minas Gerais
Belo Horizonte, MG - Brazil - 31270-901
Email: [*]tupaschoal@ufmg.br, [†]franksill@ufmg.br

***Abstract*—This paper introduces a flow for automatic fault injection into plain SystemC and SystemC/TLM (Transaction Level Modeling) designs. The aim of this flow is reducing the complexity of reliability analysis and fault coverage determination of complex designs. The flow enables the consideration of transitory and permanent faults, which can be applied in directed or random manner. Thereby, faults are represented as faulty values of variables, which are automatically identified during an initial phase. The implemented flow was successfully applied for all example designs provided by the official SystemC library. Further, the utilization of the algorithm for an exemplary technique for robustness enhancement, namely the Triple Modular Redundancy, could prove the viability of proposed flow.**

***Index Terms*—SystemC, TLM, fault injection, fault coverage, reliability**

I. Introduction

The internet of things (IoT), cars, smart cities and plenty of other systems spread throughout the society rely partly, if not solely, on electronic devices. These can be integrated circuits, Multiprocessor Systems or Systems-on-a-Chip. Given the constant progress of the miniaturization of transistor technologies, modeling a system is becoming more expensive and complex [1], making the design and verification tasks less trivial over time [2].

Determining system reliability under diverse operation conditions has become a hot topic to ensure overall system quality [3]. A mandatory step for this analysis is the modeling and injection of different types of faults into the design to study the system's robustness [4].

Among the several Hardware Description Languages (HDL) [5], SystemC [6] has stood out as a recent viable option to model systems in multiple levels of abstraction, allowing the project steps to be largely done in a single language, avoiding multiple translations and conversions.

The goal of this article is to propose a technique for automatic injection of faults into SystemC designs, in order to measure their reliability and to compare different design choices. Therefore, the development of fault models and adequate injection strategies, respecting the abstraction level of the design, is required.

The rest of this work is structured as follows. Sections II and III introduce the SystemC language and faults models. Section IV proposes the new flow, while section V presents implementation results. Finally, section VI concludes this work.

II. SystemC

Hardware Description Languages (HDLs) like SystemC are essential for the design of electronic components, as they describe the hardware in a programmatically manner [5]. The difference between SystemC and other languages already well known by the designers, such as VHDL and Verilog, is its close relation to C++. SystemC consists of a group of functions, macros and C++ definitions that were grouped together under a library by the Open SystemC Initiative [6]. Since SystemC is essentially C++, it is assumed that developers can learn easier, as they might have had already contact with C++, which has been out for more than 30 years [6]. Another advantage of being a library is the possibility to use existent and established tools, compilers and debuggers for C++, strengthening the development cycle.

Aside from common HDL definitions, such as introducing the concepts of clock, reset, ports and modules, SystemC also thrives on the use of abstractions [7]. An abstraction is essentially a way to reproduce a determined behavior in a simplified manner, without having to compute or mimic the real implementation. Figure 1 shows a comparison of the abstraction levels between Verilog, VHDL and SystemC, highlighting the flexibility of the latter.

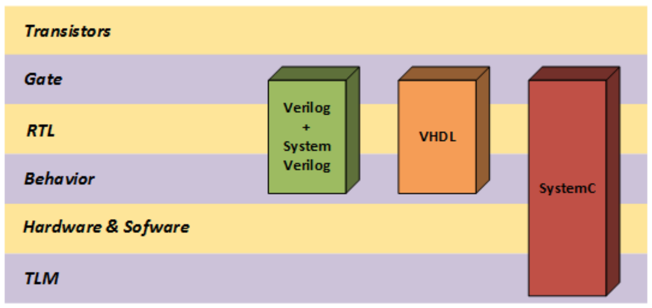

Fig. 1. Abstraction depth for Verilog, VHDL and SystemC languages.

Thus, SystemC allows the description of both software and hardware within the same language [7], blurring the line of what can be fabricated or just simulated. By doing so, it allows the designer to program a module with the desired functionality and verify if it is working properly without necessarily, having to define clock and reset domains, worry about delays nor placement and routing.

One of the efforts to speed up and improve the designing process by using a different abstraction level has been accomplished with the popularization of the Transaction Level Modeling (TLM) [8]. According to its definition, one can design a specific intellectual property (IP) that executes a desired function while using the communication interfaces described by the TLM. To connect this IP to another one, it is just necessary to ensure that both have included the interfaces as described.

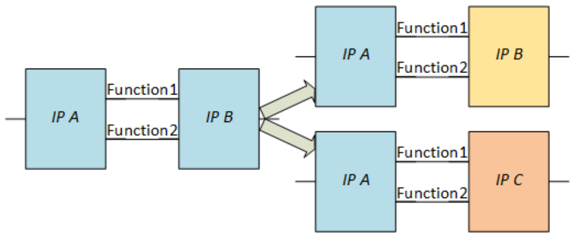

Fig. 2. TLM application in the reuse of IPs, which eliminates the need to redesign interfaces.

By standardizing the interface between modules, TLM allows easy swapping of IPs, enabling quicker and thorougher tests of different implementations [9]. Figure 2 exemplifies a case where IPs A and B are interconnected using standard communication channels, which would, in turn, allow B to be swapped by C, for example, with minimal coding.

III. Faults

A fault occurs when a system has its data altered temporarily or permanently, due to external influences or because of a design bug. Both cases are pictured in figure 3. The occurrence of a fault does not imply that the system will undergo an erratic operation, as it might have been built to withstand some faults, a characteristic known as fault tolerance.

A. Fault models

It is required to model a fault to enable its injection into a system, as it might be very particular to the elaborated architecture. Common fault models that are applicable to most designs are [10]:

- Stuck-At: A value, typically a bit, is fixed and stops changing throughout the execution of the systems' routine.
- Short and open circuits: model a faulty transistor that might greatly impact the systems' desired function.
- Delay: introduce additional delay into the propagation of a signal.

B. Fault injections

To stimulate the design into a faulty operation, it is required to inject faults in a predictable way. This can be done physically or virtually. The first would require an operating and undamaged circuit to have a signal path hit by, for example, a focused ion beam (FIB), in order to damage the circuit, creating a short or open circuit [11]. The latter requires changing the design through simulations to sabotage its modules. Examples of virtually injected faults include saboteurs [2],

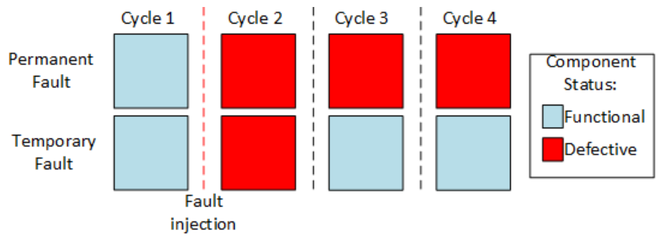

Fig. 3. Flow demonstrating the behavioral difference that distinguishes permanent and temporary faults after an injection.

which alter the communication between modules, and mutants [1], which are a faulty version of a module.

Fault injections are mostly done using simulations, which are cheaper, simpler and faster than damaging and probing real circuits. Thereby, the faults can be injected directly during simulation or indirectly before simulation. The former solution profits from faster execution times, as no recompilation is required. However, alterations of the SystemC Kernel are mandatory, which strongly reduces the applicability of the approach [12].

IV. Automatic Fault Injection

This section presents the flow for automatic fault injection into SystemC designs and the developed tool, which was realized in Python. The choice of Python was supported by its ease of use, along with the enablement of rapid prototyping and facilitated handling of file operations.

A. Fault Injection Strategy

The proposed is based on indirect fault injection, which offers a good tradeoff between practicability and desired analysis. Thereby, before each simulation the design is modified and recompiled. It should be noted that in case of complex designs the time for exhaustive simulations, as required for fault analysis, is considerable lower than the time for compilation.

During an initial phase, the design is read, viable signals are identified and the desired injection strategy is applied (see figure 4).

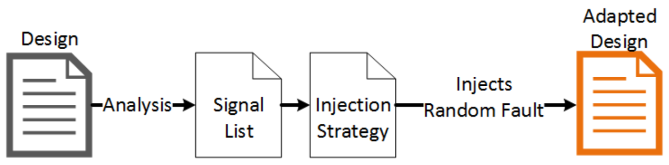

Fig. 4. Initial Steps

B. Fault Injection Candidates

An essential requirement of fault injecting is finding suitable victims, as summarized in subsection III-B. As SystemC is an extension of C++, the method chosen was to attempt to identify and single out every variable suitable for injection within the design files. This approach allows the injection to be hybridized as it might work as a saboteur in case of variables that interconnect modules or as a mutant if the variable is inside the component.

The identification of suitable variables is done via the use of regular expressions. Thereby, search patterns are defined that are capable of isolating occurrences of a specific sequence of characters within the analyzed text, i.e. the program code.

Algorithm 1. Regular expression to find variables declared within a SystemC module

```
regEx = re.compile(
#Const Identification
'(const |[^A–Za–z])'
#C++ Types
'(int|float|short|char|bool|double'
#SystemC Types
'|sc_(?:bit|logic|int|uint|bigint|biguint))'
#Templates
'(?:\<\w*\>)?'
#Ignores *&' '
'(?:[ \*&] *\*{0,2}&{0,1} *)'
#Variable Name
'([A–Z_a–z]\w*)'
#Ending in: =);,[]
'[ ,;\)\[\]]')
```

Algorithm 1 shows the regular expression used for identifying the variables in a SystemC module. It discards the ones preceded by ”const”, as they cannot have their value changed and then identify the type as either from native C++ or SystemC. Then it skips structures such as templates, references and cast operators to obtain the variable name.

Algorithm 2. Regular expression to find payload variables declared within a SystemC environment with TLM.

```
regEx = re.compile(
#Finds operation
'(?:−>)'
#Identifies transport type
'(?:nb_transport_[bf]w|(?:b_transport))'
#Ignores delimiters
' *\( *\* *'
#Captures payload
'([A–Z_a–z]\w*)'
#Ends capture
'[$ ,\n]')
```

Algorithm 2, on the other hand, identifies the names of TLM payloads by delimiting what commonly precedes and follows a payload variable, the transport operation call and the delimiters, respectively. Hence, it is possible to isolate the payload name to be replaced upon the injection of a fault.

C. Fault Generation

Once the variables have been identified, faults are injected into the design. The execution is as follows:

1) Insert probability control algorithm: A utility algorithm, named randomBool, is inserted in the design's source code, which allows the definition of the fault occurrence probability. Controlling this probability will determine the fault type, as depicted on figure 3.
2) Insert invalid payload initialization: If the design is using TLM, an invalid payload is created and initialized at the beginning of the design.
3) Insert faulty variable assignment: Whether the fault is within pure SystemC or TLM, the variable found by the algorithms 1 and 2 is replaced by the ternary construction pictured in algorithm 3, which will assign the faulty value to the variable with the probability chosen in step 1. The faulty value is a random value generated by the tool that agrees with the type identified.

Algorithm 3. New variable value assignment, in ternary construction to allow the probabilistic fault injection.

```
// randomBool() is executed and if it
// returns true, the fault is injected,
// otherwise nothing happens.
variable = randomBool() ? injectedValue
                        : variable
```

D. Fault Injection

Following the steps described in IV-B and IV-C, the fault will be injected in a modified design file, and the tool will try to compile and then execute it. Within this context, three possible outcomes might emerge:

1) Compilation failure
2) Unsuccessful execution or timeout
3) Normal Execution

Only the first scenario is considered a tool failure, as it managed to break the compilation of a working design when injecting the fault. Consequently, the tool was not able to inject a fault into an eligible variable, reducing the fault injection coverage.

V. Results

This section presents the results obtained during application of the proposed flow for example designs.

A. Example Designs

The flow for automatic fault injection was applied for all designs provided by the SystemC and TLM libraries. There are a total of 20 SystemC projects and 11 SystemC/TLM designs, which range from simple, such as adders, to complex systems, like a RISC CPU or RSA encryption algorithm. In each design, the flow tried to inject a failure in every candidate variable, being an exhaustive injection flow. The goal was to widen the tool test environment to encompass plenty of scenarios.

The compiled results are pictured in the figure 5. It is possible to perceive that the implemented tool was mostly successful, as the sum of failed executions and normal executions exceeded the number of failed compilations.

One of the greatest causes of compilation failure within the SystemC designs was the mismatch of the injection value and the variable when the latter was an array, and made the compiler expect a pointer, instead of a static variable. In contrast, TLM had a much lower compilation rate, as the injection points are expected to vary less and be more predictable, hence the regular expression shown in algorithm 2. On the other hand, TLM exhibited an expressive rate of failed execution and this is due to the easiness at which defective payloads can lead the design to read improper memory addresses.

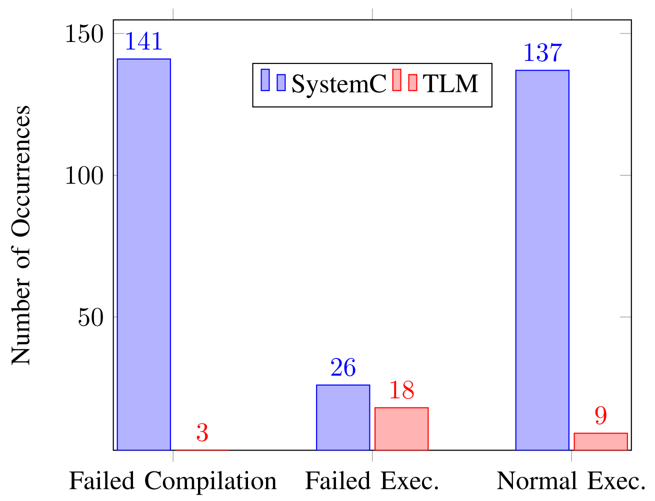

Fig. 5. Outcome of the fault injections in all designs provided with the SystemC library. The number of occurrences show variables that might have been manipulated, depending on the outcome shown on the horizontal axis.

B. TMR Example

One application of the automatic fault injection tool can be exemplified with the study of simple circuits against their version with Triple Modular Redundancy (TMR). TMR is the replication of the circuit into three identical ones, where their outputs are compared so that the prevailing output will be equal to the majority of the outputs of the individual circuits [13].

The example circuit uses a TMR arbiter to decide the correct result of a sum performed by three 64-bit adders. Faults were injected in all the adders, as well as the TMR module, with p_a being the probability of failure within an adder and p_t the probability within the TMR component. Each design configuration was simulated a total of ten thousand times, an empirical value that allowed convergence of the failure rate.

TABLE I
PERCENTAGE OF INCORRECT RESULTS UPON FAULT INJECTION IN EQUIVALENT SYSTEMS WITH AND WITHOUT TMR.

Without TMR		**With TMR**				
		$p_t = 0$		$p_a = \frac{1}{100}$		
$p_a = \frac{1}{10}$	$p_a = \frac{1}{100}$	$p_a = \frac{1}{10}$	$p_a = \frac{1}{100}$	$p_t = \frac{1}{100}$	$p_t = \frac{1}{500}$	$p_t = \frac{1}{1000}$
9.00%	0.79%	2.69%	0.02%	0.77%	0.15%	0.09%

Table I shows that the automatic fault injection tool can be used as a way to compare the reliability of two different design choices. It follows, that the version with TMR had better fault tolerance than the version without, with improvements ranging from 3 to 40 times less failures. The design with TMR was superior even when the component responsible for doing it had a probability to fail comparable with the adders' failure probability. This is a highly unlikely scenario as in reality the TMR circuit is simpler than the components whose faults it mitigates, i.e. being less prone to failing than adders.

VI. Conclusion

SystemC is predicted to grow in the following decades, as industries move towards the unification of design and verification programming languages.

Assessing the reliability of a component or design is fundamental to the development of a successful system, but exposing it to faults might be a difficult and slow process. Hence, there is the need for an automatic fault injection tool.

The tool developed was able to inject faults in both SystemC and TLM by changing the assignments of a component's variable, with minimal user input, typically only the design path. At least one candidate variable was found in each reference design and although some of them had high rates of compilation failures, overall the number of executions outnumbered them.

Finally, it was demonstrated that the tool can be used as a way of comparing different design architectures, as injecting faults with different probabilities of failure can enable the assessment of a system's reliability.

Acknowledgment

The authors would like to thank CNPq, FAPEMIG, UFMG/PRPq for financial support.

References

[1] C. Bolchini, A. Miele, and D. Sciuto, "Fault models and injection strategies in SystemC specifications," *Proceedings - 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools, DSD 2008*, pp. 88–95, 2008.
[2] S. Misera, H. T. Vierhaus, L. Breitenfeld, and A. Sieber, "A mixed language fault simulation of VHDL and SystemC," *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, DSD 2006*, pp. 275–279, 2006.
[3] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *Computer*, vol. 39, no. 1, pp. 118–120, jan 2006.
[4] A. Avizienis, H. Kopetz, and J. C. Laprie, *The Evolution of Fault-Tolerant Computing: In the Honor of William C. Carter*, ser. Dependable Computing and Fault-Tolerant Systems. Springer Vienna, 2012. [Online]. Available: https://books.google.com.br/books?id=dm6rCAAAQBAJ
[5] R. W. Hartenstein, *Hardware description languages*. North-Holland, 1987.
[6] Accellera Systems Initiative, "About SystemC," 2015.
[7] J. Bhasker, *A SystemC primer*, 1st ed. Star Galaxy Publishing, 2002. [Online]. Available: h ttp://www.stargalaxypub.com
[8] Accelera Systems Initiative, "SystemC TLM (Transaction-level Modeling) Working Group," 2012. [Online]. Available: http://accellera.org/activities/working-groups/systemc-tlm
[9] European Space Agency, "System-Level Modeling in SystemC." [Online]. Available: http://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/System-Level_Modeling_in_SystemC
[10] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. Boston: Pearson Education, 2011. [Online]. Available: https://books.google.com.br/books?id=4SYrAAAAQBAJ
[11] R. C. Aitken, "Finding defects with fault models," in *Test Conference, 1995. Proceedings., International*, oct 1995, pp. 498–505.
[12] 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.
[13] A. Majumdar and S. B. K. Vrudhula, "Techniques for estimating test length under random test," *Journal of Electronic Testing*, vol. 5, no. 2-3, pp. 285–297, 1994.