A Domain-specific Language for Automated Fault Injection in SystemC Models

Douglas Lohmann, Alexis Huf, Djones Lettnin, Frank Siqueira and José Luís Güntzel
Federal University of Santa Catarina - Florianópolis, Brazil
{lohmann.d,alexis.huf}@posgrad.ufsc.br. and {djones.lettnin,frank.siqueira,j.guntzel}@ufsc.br

***Abstract*—With the evolution of technology, electronic systems have become significantly more complex. As a consequence, design and verification of these systems evolved notably. Fault injection is a dependability evaluation technique that is strongly recommend during the verification step. Although there are a number of tools capable of injecting faults, many of them do not have a simple fault model description language and require considerable manual effort. In this paper, we propose a SystemC template metaprogrammed Domain-Specific Language (DSL) integrated with Universal Verification Methodology (UVM) to describe formal fault models that requires neither specific compilers nor code preprocessing tools. Unlike current approaches for fault injection, there is no need to create fault injection environment manually or to describe the system in an XML format. We evaluate our approach in terms of readability and effort required from a designer to describe a fault injection test. Our case study illustrates how the DSL helps designers to create fault models in SystemC, decreasing programming effort and taking advantage of SystemC/C++ expressiveness.**

***Keywords*—Fault injection, SystemC, Universal Verification Methodology.**

I. Introduction

Over the last decades, electronic systems have become more and more complex, due to the advances in chip fabrication technology and the social needs for technologies development. Given that embedded systems play an important role in our daily life, it is fundamental that those systems are reliable otherwise some system failures can eventually threaten lives. The need for reliability and the growing complexity of embedded systems introduce new challenges, mainly in the verification process, to ensure the correct behavior of the design.

In order to make system verification easier, engineers introduced more layers of abstraction in the design flow, facilitating description and testing at the system level. As a possible solution, electronic systems are described in a high-level language, such as C++ or Java, and then the system is translated to a hardware description language, such as VHDL or Verilog. The drawbacks of this approach are the manual effort for translation and the risk of introducing new errors during this step. SystemC language was created to overcome those issues by enabling all levels of design modeling, for both hardware and software, by using the same language.

In addition, methodologies to create reusable Testbenches and techniques to improve the system's dependability can be coupled to optimize the verification process. The most used Testbench methodology is the Universal Verification Methodology (UVM), which sets rules and guidelines for enhancing Testbench development and simulation execution through code reusability, modular Testbenches, stimulus generation and transaction-level communication between the test ambient and the Design Under Test (DUT), among other qualities [1]. One method for improving dependability evaluation is the deliberate injection of faults into a system, followed by an evaluation of how well the system was able to deal with the faulty behavior of some components. The fault injection technique can also be integrated with the UVM Testbench, as both are used for system verification and validation.

Faults are described in a fault model, containing the fault type, fault trigger, and the fault location. One issue on the existing fault tools for SystemC is how to describe the fault model. Many fault injection tools are based on the Extensible Markup Language (XML) or use a runtime console. Despite the number of fault injection methods using it, the XML description of fault models is verbose, making it difficult to setup system verification under fault injection. To overcome this, in this paper we propose a Domain-Specific Language (DSL) for fault model description. To that end, we create a test environment based on UVM extensions to allow fault injection at SystemC designs, as done in [2], but with the DSL improvements. The DSL was implemented with C++ template metaprogramming and connected to the UVM fault test environment.

Approaches that employ manually interactions are challenging to perform batch tests and replicate experiments. Approaches that employ XML have a high entry barrier caused by its verbosity and the distance of XML from the well-established SystemC design workflow. Each tool also requires a specific structure for the description of fault models, that must be learned by the programmer. Finally, API-based approaches avoid the limitations of XML but still yield complex setup code that is difficult to maintain.

The remainder of the paper is organized as follows: Section II presents our SystemC DSL for fault models description. Section III shows the experimental results. Section IV analyzes related works, and Section V presents the conclusions and limitations of our approach.

II. DSL for Fault Model Description

The fault injection environment builds upon the UVM, adding an Event-Condition-Action (ECA) engine and a template metaprogrammed DSL for fault model description. The

 DOI: 10.1109/icecs.2018.8617838

ECA engine receives fault models from the test engineer and operationalizes fault injection. A fault model consists of a fault trigger (i.e., a condition that when it becomes true, triggers fault injection), a fault location (i.e., wherein the DUT the fault will be injected) and a fault type (i.e, what will be interference). The ECA engine splits the fault trigger into a set of event sources and a boolean condition yielding, respectively, the event and condition elements of ECA. The fault location and fault type are given by a single fault injection component, that the ECA engine stores as the action element.

Given that the UVM, the ECA engine and supporting event monitors are independent of the DUT, the specification of the fault model remains the most complex task for the test engineer. The main goal of the DSL is to build expressions mixing C++ values or variables with SystemC variables or objects (e.g., signals). The DSL is implemented using templates and retains the syntax of C++ expressions. As for semantics, the code generated by the C++ compiler for a DSL expression only creates a data structure for subsequent introspection and evaluation. The atomic elements of the DSL, whose presence makes the C++ compiler treat the whole expression as a DSL expression, are `var` and `evt`. A `var` object stores the name of a DUT object registered in the UVM configuration database. An `evt` object is similar to a `var` object, but refers to the default event of SystemC ports and signals). To the built-in C++ operators `var` behaves like the referred object and `evt` behaves as a boolean (true after the event occurs).

In addition to `var`, `evt` and the built-in C++ operators, the DSL includes other helper constructs. The `cap(x)` function allows the DSL expressions to bind to a C++ variable `x` by reference, instead of copying its value. The `unif(a, b)` function represents a random value uniformly distributed between `a` and `b`. The `call(f, a1, ...)` function represents the return of function `f` if called with arguments `a1, ...`, which may themselves be a DSL expression. This provides a escape hatch for any situation where complex or stateful calculations are required as part of the DSL (e.g., coordinate transformations or Finite Impulse filters).

III. Experimental Results and Discussion

This section evaluates the approach presented in Section II with respect to expressiveness and ease of use. In addition, we show a fault injection case study, in which we successfully applied our technique to build a fault-tolerant matrix multiplication design.

A. Comparing the DSL to existing approaches

The evaluation employs three scenarios of fault injection in order of increasing complexity. Three fault model description techniques are applied to each of these scenarios. The first technique is the XML-based one proposed in [3]. This was the only approach for which we could obtain some code describing fault injection. However, the chosen scenarios are limited by the absence of the full language specification and the actual tool. The second technique is the one described in our previous work [2] with an additional ECA implementation but lacks the DSL. The third technique is the DSL proposed in this paper.

Scenario A consists of data modification. Listing 1, adapted from [3], specifies one such scenario. In this example, one byte, starting at position 200, is modified. As the modification is a `or` mask, the effect is an assignment of that one byte.

Listing 1. Scenario A: XML fault model adapted from [3]

```
<data>
  <transfer start pos="200" length="1">
    <or mask="0xff"/>
  </transfer>
</data>
```

Listing 2 shows the same fault model description using the proposed fault injection technique using the ECA, without the DSL. As discussed in Section II, our fault injection technique is based on [2], where the authors use the *uvm_config_db* to access DUT components. Therefore, before the fault condition is defined, the objects used in the description must be in the UVM database. Any accessible DUT component from the Testbench can be stored in the UVM config database, such as signals, ports, and other SystemC types.

Listing 2. Scenario A: fault model without DSL

```
bool truth = true;
engine->register_fault_condition(new uvm_var_ct_tpl<bool>(1),
   new uvm_fault_set(new uvm_var_tpl<bool>("tgt", "*"),
      &truth, 0, 1));
```

The size of the code in Listing 2 hinders its readability. The proposed DSL generates the same effects but with a more compact and intuitive syntax. Listing 3 shows the fault model description from scenario A expressed in the DSL. As the condition `true` alone is not a DSL expression, it requires wrapping. The fault type is to set a char value on a variable `tgt` of type `char`. In general, any DSL expression, in addition to C++ values can be used as argument of `set()`. For example, `var<char>("x") & var<char>("y")`.

Listing 3. Scenario A: fault model with the proposed DSL

```
*engine << fm(cnst(true), var<char>("tgt").set('\xff'));
```

The *register_fault_condition* ECA *engine* method is used to register a fault model at the Testbench. To make the DSL more compact, the << operator is overloaded to be used in place of the aforementioned method. As is the case when this operator is used with C++ output streams, fault models can be repeatedly streamed into the ECA engine. A helper function, `fm` (from "fault model") is defined to group a fault trigger (as the first argument) and a fault injector (as the second argument), both described using the DSL.

Scenario B presents a more complex fault condition, introducing a probability. In this scenario, the variable is set in a certain percentage of its triggers occurrences. To describe this type of conditions we use the `unif` function, that creates a uniform random distribution. Listing 4 uses random value in such distribution to build an expression whose value is `true` 90% of the times the expression is evaluated.

Listing 4. Scenario B (probabilistic): DSL description

```
*engine << fm(unif(1,10) <= 9, var<char>("tgt").set('\xff'));
```

Faults with probability can also be created using the XML approach presented in [3]. To do that, the XML attributes

 DOI: 10.1109/icecs.2018.8617838

`random=1` and `percentage=90`, must be added to the `transfer` tag at the second line of Listing 1.

Scenario C sets the value of a variable `tgt` to the average of the last 3 values observed for the signal `m_out1`, Listing 5 shows the DSL fault mode description for this scenario. This scenario relies on the `call` helper to compute the moving average with a user-defined function `mov_avg`. The moving average is updated always every time the `m_out1` signal changes. In the code, `mov_avg` is a template function that computes the moving average from a history vector pointer (`&hist`), a window size (`3`) and a sample value (`m_out_1` current value). This scenario is not possible in [3]. Furthermore, the ability to call user-defined stateful functions as part of fault injection is not present in any fault injection tool, to the best of our knowledge.

Listing 5. SCENARIO C (MOVING AVERAGE): DSL DESCRIPTION

```
*engine << fm(evt<sc_signal<sc_int<32> >*>("m_out1"),
   var<sc_int<32> >("tgt").set(call(&mov_avg<sc_int<32> >,
      cnst(&hist), cnst(3), var<sc_int<32> >("m_out1"))));
```

There are no absolutely fair metrics to compare XML and C++, as the languages do not share common elements such as statements, declarations or expressions. Comparing line numbers is also not fair as C++ lines are often longer than XML ones. A reasonable metric is character count (ignoring all optional spaces and line breaks), which is shown in Table I for each scenario and technique combination. In all scenarios, the DSL shows the smallest value. The main advantages of the DSL, however, become clear in a qualitative comparison. First, expressions (for fault trigger or fault type specification) are more compact and readable in C++, especially for expressions with more than a single operator. In XML, expressions must be represented in the form of a tree. In contrast, the DSL automates the generation of a similar tree implicitly. Second, `var` and `evt` objects, as well as whole DSL expressions, can be stored on local variables and reused across several fault models. Third, `call` and the ability to mix C++ and DUT objects in the fault model allows for greater flexibility, and allow the test engineer to program highly specific fault models which cannot be foreseen by fault injection tool designers.

B. Matrix multiplier with fault tolerance mechanism

In this section, we perform a fault injection test in a triple matrix redundancy (TMR) multiplication example. For that, we develop a DUT as shown in Figure 1. The DUT receives as input two matrices, performs three independent multiplication algorithms and chooses the most frequent matrix, as a strategy for fault tolerance. The connections between the matrices and the voter are made using *sc_signal* standard communication.

Table II shows the fault model applied in our example. Each fault registered is composed of a fault trigger, a fault type and a fault location. In our example, a fixed value "*fixed n*" is set at signals connecting the matrix multipliers ("m1", "m2" and "m3") to the voter component, for each multiplier input event with a uniform probability of 33%.

TABLE I. CHARACTER COUNT PER TECHNIQUE PER SCENARIO

	Technique	Scenario A	Scenario B	Scenario C
Characters	[3]	78	104	-
	[2]	141	183	381
	DSL	55	60	159

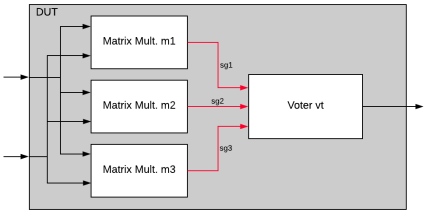

Fig. 1. DUT with fault tolerant matrix multiplication architecture.

TABLE II. FAULT MODELS

Fault trigger	Fault type	Fault location
`evt<out_m1>("m1_out",scp)` `&& unif(0,100) <0.33`	`set_value(n)`	`sg1[12]`
`evt<out_m2>("m2_out",scp)` `&& unif(0,100) <0.33`	`set_value(n)`	`sg2[12]`
`evt<out_m3>("m3_out",scp)` `&& unif(0,100) <0.33`	`set_value(n)`	`sg3[12]`

The Listing 6 demonstrates a fault condition expression for the first line of Table II fault model. The `unif` is a factory function for a random variable, and is part of the DSL.

Listing 6. REGISTERING A FAULT CONDITION

```
sc_int<16> a = 323;
*engine << fm(evt<sc_out<sc_int<16> >*>("m1_out", "*")
      && 33 > unif(0,100),
   var<sc_signal<sc_int<16> >*>("signal_m1", "*").set(cap(a)));
```

After registering the entire fault model of Table II, we run the experiment for lengths of 50, 100, 500, and 1000 simulations. Table III presents the results for those experiments. In this table, a failure is determined by a DUT output matrix (after the voter) that is different from the correct multiplication of the input matrices. A success is a correct multiplication matrix output, even when faults are injected. We note that the fault injector applied the faults, and the tolerant matrix algorithm is able to tolerate around 82% of introduced faults.

TABLE III. FAULT INJECTION RESULTS

Sim. length	Injected faults	Failures	Success
50	41	8	42
100	85	16	84
500	484	94	406
1000	1001	187	813

This case study has shown that performing fault injection using the DSL is intuitive and feasible. Many other fault models can be described using this technique with the SystemC modeling. Different kinds of triggers, fault types and locations can be combined to create more efficient fault injection campaigns.

IV. RELATED WORK

A fault injection technique based on replacing the original data and signal types is presented in [4]. For that, the authors

 DOI: 10.1109/icecs.2018.8617838

overloaded the SystemC types constructors and destructors to insert their fault types. Although this approach is less intrusive than those presented in [5], it needs source code modifications to change the constructors of data types and enable the faults. The faults are set by a Fault Injector Manager through commands. It is possible to choose fault types, locations, and set probabilities of fault occurrences. All these properties of the fault model are configured using methods of a class named *FaultInjectionPolicy*.

Another fault injection technique based on data type extensions is presented in [6]. Instead of overwriting the data types constructor at DUT level as in [4], the authors modify the SystemC kernel to change the native types of the language. However, the focus of the article was not the description of the fault model, but the fault injector introspection mechanism and its optimization in terms of execution time.

In [7] and [8], a Reflective Simulation Platform (ReSP) is presented. ReSP was implemented in Python and is based on the reflective property of this programming language. In the ReSP environment, faults can be described by the console, which is useful during debug. To conduct fault injection campaigns or perform many runs on the same architecture, it is necessary to describe the fault models in an XML file, in the same way as other fault injectors. Neither [7] nor [8] presented the XML syntax for fault model description or example descriptions. The XML description introduces repeatability of fault injection testing, but as was the case in [3], it is more verbose than a fault model description contained in SystemC itself. Another example of work using XML for fault model description is [9]. In their approach, the authors generate SystemC faulty components from XML descriptions.

Our fault injector fully integrated with UVM is a user-friendly formal fault model description tool that to allows engineers to write fault models in a language that supports hardware and software development, differently of [7] and [8]. Besides that, the faults are described using the C++ operators to increase the expressiveness, while the XML-based works have an unnatural description from the perspective of UVM. Furthermore, such approaches are more verbose than the DSL to express the same fault model, as showed in Section III while comparing our work with [3].

V. Conclusion

In this paper, we presented a Domain-Specific Language (DSL) in order to create compact and readable fault model descriptions. We showed the architecture created to support the fault injection environment and the fault model description method. Our approach is based on extensions of the Universal Verification Methodology (UVM) to create an hybrid environment for verification and injection of faults with a C++ DSL for fault model description. Case studies conducted in this paper demonstrate that the DSL is less verbose and more readable than the other XML-based approaches. Notwithstanding, we also evaluated the same fault injector framework with and without the DSL, and demonstrated that the DSL is more usable than directly creating C++ objects that describe a fault model.

We validated our fault injection technique with a triple-redundant matrix multiplier fault tolerant design, developed in SystemC at a high level of abstraction. The performed tests demonstrate the intuitiveness and effectiveness of our approach. Template metaprogramming is widely known to provide additional strain to the compiler. Another possible limitation is that our apporoach ueses polymorphism relying on heap memory (new operator). However, as the Testbench typically runs on high-end PC systems instead of embedded systems, these characteristics have negligible effect. Furthermore, our approach requires no changes to the DUT, and therefore the same code that yields the synthesized final system can be tested on the Testbench.

The DSL improves the fault injection tests, providing a better way to describe the fault model. This approach contributes to improving system dependability evaluation and system verification. Future works consist in improvements at the automation of the DUT variables insertion in the UVM database, and the ability to verify at compile time if all variables used in a DSL expression were inserted in the database.

Acknowledgment

Alexis Huf receives a scholarship from FAPESC/CAPES and Douglas Lohmann received a scholarship from CAPES.

References

[1] S. Rosenberg and M. Kathleen, *A Practical Guide to Adopting the Universal Verification Methodology (UVM) Second Edition*. Cadence Design Systems, Inc., 2013.

[2] D. Lohmann, F. Maziero, E. J. Santos Jr, and D. Lettnin, "Extending universal verification methodology with fault injection capabilities," in *9th Latin American Symposium on Circuits & Systems (LASCAS)*. IEEE, 2018.

[3] M. Michael, D. Groβe, and R. Drechsler, "Analyzing dependability measures at the electronic system level," in *Specification and Design Languages (FDL), 2011 Forum on*. IEEE, 2011, pp. 1–8.

[4] 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.

[5] 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.

[6] W. Lu and M. Radetzki, "Concurrent and comparative fault simulation in systemc and its application in robustness evaluation," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 115–128, 2013.

[7] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto, "Resp: A non-intrusive transaction-level reflective mpsoc simulation platform for design space exploration," *inProc. ASPDAC*, pp. 673–678, 2008.

[8] C. Bolchini, A. Miele, and D. Sciuto, "Fault models and injection strategies in systemc specifications," in *Digital System Design Architectures, Methods and Tools, 2008. DSD'08. 11th EUROMICRO Conference on*. IEEE, 2008, pp. 88–95.

[9] W. Yan, D. Fontaine, J. A. Chandy, and L. Michel, "A design flow with integrated verification of requirements and faults in safety-critical systems," in *System of Systems Engineering Conference (SoSE), 2017 12th*. IEEE, 2017, pp. 1–6.