SystemC-based Minimum Intrusive Fault Injection Technique with Improved Fault Representation

Rishad Ahmed Shafik, Paul Rosinger, Bashir M. Al-Hashimi
School of ECS, University of Southampton, Southampton, UK, SO17 1BJ
Email: {*ras06r, pmr, bmah*}@*ecs.soton.ac.uk*

Abstract

In this paper, we propose a new SystemC-based fault injection technique that has improved fault representation in visible and on-the-fly data and signal registers. The technique is minimum intrusive since it only requires replacing the original data or signal types to fault injection enabler types. We compare the proposed simulation technique with recently reported SystemC-based techniques and show that our technique has fast simulation speed, better fault representation, while maintaining simplicity and minimum intrusion. We demonstrate fault injection capabilities in a behavioural SystemC description of MPEG-2 decoder using proposed technique and show that up to 98.9% *fault representation within data and signal registers can be achieved.*

1 Introduction

Electronic systems reliability is a growing concern in deep sub-micron, in particular due to increased errors due to higher integration and packing density. SystemC is high-level design language with potential improvements in design productivity by allowing the designer to operate at higher- or mixed-levels of abstraction than available with traditional RT-level synthesis tools, such as VHDL or Verilog. An important aspect in the development of such electronic systems is how to validate the feasibility of fault-robust design in an early phase to reduce the re-design cost. Existing fault injection (FI) tools have been mainly VHDL based, most of these have been limited to RT-levels [1]. For large and complex systems, fault injection using these techniques require time-consuming re-design and validations, which can be greatly simplified using SystemC [2, 3]. With growing interest in SystemC-based development, simple, minimum intrusive, and fast fault injection technique with fault access and representation in all visible registers is much needed.

Existing SystemC-based fault injection techniques reported in [4, 5, 6] are saboteur and mutant based, which have poor fault access and representation in different data and signal registers and also has highly intrusive and time consuming re-design technique for fault injection. Recently reported techniques in [3, 2] have poor fault access for different data and signal types due to signal type restrictions. In this paper, we present a minimum intrusive and fast SystemC-based FI technique implemented in a prototype simulator that works by replacing data and signal types to equivalent FI enabler types. The proposed FI technique, thus, achieves very high fault access and representation for all visible and on-the-fly registers. Similar approach to simulation modeling for design space exploration was reported in [7].

2 Existing Fault Injection Techniques

Existing simulated FI techniques are based on saboteurs, mutants or simulation command-based approach [1]. SystemC-based FI techniques using emulation approach (such as [8]) is outside the scope of this paper due to our focus on simulated FI approach. In the following, existing simulated FI techniques are described.

2.1 Saboteurs and Mutants

Saboteur is a special component added between signal drivers and receivers to alter their value or timing characteristics when a fault is injected [1]. Due to signal based approach, saboteurs have no access to data registers, such as variable and on-the-fly registers [9]. Also, for designs with large number of signals, saboteurs have limited practical application. To ease such re-design,VHDL-based saboteur libraries are proposed in [1] and automatic saboteur insertion technique was proposed in [10]. However, for multi-level abstractions, such techniques are not practical due to poor fault access and highly intrusive design modifications. Saboteur-based FI using SystemC is presented in [4]

and distributed saboteur-like fault injection controllers are shown in [5].

On the other hand, a *mutant* is a component description that replaces another component description. Mutants can be accomplished either by adding saboteurs to structural or behavioural component descriptions, or by mutating structural component descriptions or behavioural components to achieve complex and detailed fault models [1]. Similar to saboteurs, mutants require time consuming re-writing or conversions, and therefore, mutant-based re-design can become complex for large designs. SystemC-based external signal manipulation for fault simulation has been carried out in [6], showing attack based fault scenarios and using fault insertion models more like mutant based designs.

2.2 Simulation Command

Simulation command-based approach, which is employed in this paper, is a powerful and minimum intrusive way to inject faults via variable or signal manipulation [1]. Variable manipulation technique works by altering values of variables, while signal manipulation technique works by altering the value of signals in the system description. Controlling the observation time different faults can be manifested using such techniques. Signal manipulation technique in SystemC requires re-wiring abilities. Although, new SystemC version 2.2 allows ports to be open during runtime, unlike previous versions, dynamic re-wiring is still not possible.

Recently, SystemC simulation command-based approach have been reported by [2, 3], which employ the introduction of extra values within SystemC type *sc_logic* to control stuck-at-faults, which are *sc_logic_A* meaning a stuck-at-1, *sc_logic_B* meaning a stuck-at-0, and *sc_logic_R* meaning *reset* for stuck-at faults. However, in behavioural models involving different variable and signal types, restriction to only *sc_logic* or its array type *sc_lv<N>* may require programming effort and conversions. Often such conversions are not feasible for primitive types, such as *float* and SystemC types, such as *sc_fixed<..>*. Hence, simulation command-based approaches [2, 3] for FI have limited access to fault injection for such systems.

3 Proposed Fault Injection Technique

The proposed fault insertion technique has three major components as shown in a prototype simulator in Fig. 1: the database of possible fault locations, fault policy manager (FPM) and finally and the fault injection manager (FIM). The system clock is connected to the FIM (Fig. 1) to enable fault injection over system timespace. Brief description of each component follow.

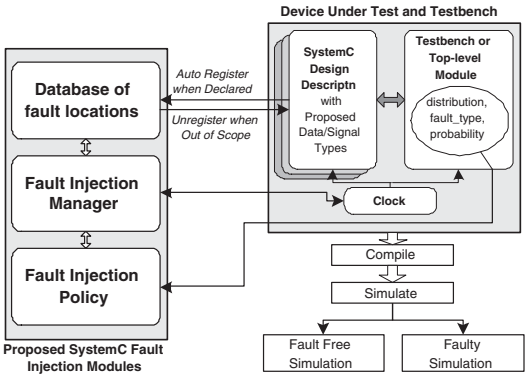

Figure 1. Proposed fault injection technique in a prototype simulator

3.1 Database of Possible Fault Locations

The proposed FI technique works by automated creation and update of database of possible fault locations, initiated by replacing the original data and signal register types to the FI enabler types shown in Table 3.1 (most commonly used C++/SystemC types shown but other types are also implemented). The FI enabler types (Table 3.1) have careful and transparent implementation of their original types such that their functions are kept intact. The FI enabler types manages fault locations' database by overloading the initialisers (constructors) and de-initialisers (destructors) in different scopes in the SystemC descriptions (Fig. 1). The database contains type, fault and size specific information with thread-safe update such that no mutually inclusive updates are allowed. The SystemC definitions of proposed *Reg<primitive type>* and *IntReg<N>*, which are equivalent to primitive types and SystemC *sc_int<N>* type, are shown in Fig. 2. As shown in Fig. 2(a), for *Reg<primitive type>*, the constructor function registers the address pointer of the original type to a centralised list by calling its member function *registerInsert(..)*. Similarly, in Fig. 2(b), each of the overloaded constructors of *IntReg<N>* registers address pointers of value-holder variable *m_val*. Both type implementations have destructors that have function call to

Original types	Fault injection enabler types
primitive type (int, bool etc.)	Reg<primitive type>
sc_logic	LogicReg
sc_int<N>	IntReg<N>
sc_bigint<N>	BigIntReg<N>
sc_lv<N>	LogicVectorReg<N>
sc_bv<N>	BitVectorReg<N>

Table 1. SystemC/C++ types and corresp. fault injection enabler types

```
//Reg.h
#define BYTE 8 //byte->bits
#define size(x) ((int) (sizeof(x) * BYTE))
template <typename T>
struct Reg{
  T reg;
  //Constructor
  Reg(T _reg = 0) {
    reg = _reg;
    FIMgr::getInstance().registerInsert
    ((void *) &reg, size(reg), DATA_TYPE);
  }

  //Destructor
  ~Reg() {
    FIMgr::getInstance().registerDelete
    ((void *) &reg);
  }
.....
};
```

(a)

```
//IntReg.h
template <int W>
class RegInt : public sc_int_base{
  ....
  //Constructors..
  IntReg():sc_int_base(W){
    FIMgr::getInstance().registerInsert
      ((void *) &m_val, W, DATA_TYPE);
  }
  IntReg(int_type v):sc_int_base(v, W){
    FIMgr::getInstance().registerInsert
      ((void *) &m_val, W, DATA_TYPE);
  }
  ......
  //Destructor
  ~IntReg(){
    FIMgr::getInstance().registerDelete
      ((void *) &m_val);
    }
  ......
};
```

(b)

Figure 2. SystemC definition of (a) *Reg<primitive type>* **as a replacement of primitive types, (b)** *IntReg<N>* **as replacement of SystemC** *sc_int<N>* **type**

registerDelete().. to de-register from the database, when the scopes of these types are expired (Fig. 2).

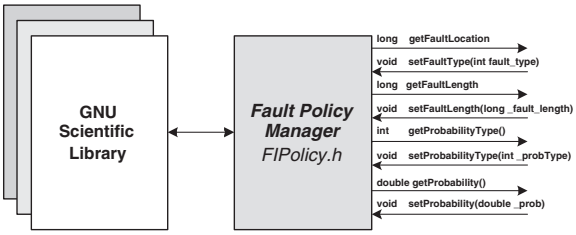

Figure 3. The fault policy manager

3.2 Fault Policy Manager

Fault policy manager (FPM) represents the second component of the proposed FI technique (Fig 1) and controls the location and probability of fault injection from the top-level module or testbench. In Fig. 3, overview of FPM, *FIPolicy*, is shown with some member functions. While *setProbability(..)* and *setProbabilityType(..)* set the probability and its associated type for the fault injection, *getFaultLength(..)* and *getFaultLocation(..)* return to the FIM the associated search length and fault location within the available bit space within the list of data or signal registers as possible fault locations. Different fault types (stuck-at, SEU, delay or indetermination) can be set by *setFaultType(..)* and current fault type can be returned by *getFaultType(..)*. Random number generation using different probability distribution is managed with the help of GNU scientific library.

```
....//Header file inclusions
class FIManager: public sc_module{
  .....
  SC_CTOR(FIManager){
    .....
    SC_CTHREAD(insert_faults, clk.pos();
  }
  void insert_faults(){
    .....
    list<RegisterElement>::iterator listIterator;
    long loc_counter;
    while(true){
      wait();
      fault_length = currentPolicy.getFaultLength();
      fault_loc = currentPolicy.getFaultLocation();
      loc_counter = 0;
      for(listIterator = mylist.begin(); listIterator != mylist.end();){
        //Insert fault or clear it at fault_loc.....
        loc_counter+=(*listIterator).size_bits; listIterator++;
        if(loc_counter < fault_length && listIterator == mylist.end())
          {wait(); listIterator = mylist.begin();}
      }
    }
  }
  .....
  FIPolicy currentPolicy; list<RegisterElement> mylist;
};
typedef Singleton<FIManager> FIMgr;
```

Figure 4. SystemC description showing fault injection method for the FIM

3.3 Fault Injection Manager

Fault injection manager (FIM) is the third and main module that interfaces with the FPM to inject faults within the database of possible fault locations. In order to sample fault insertion time into discrete fault probability in bitspace, system clock of the top-level module/testbench is connected to the FIM (for homogeneous, single-clock systems) by SystemC statement

`FIMgr::getInstance().clk(top_level_clock);.`

SystemC description of the FIM module, *FIManager*, is shown in Fig. 4 with main FI function *insert_faults*. For single or concurrent multiple fault injection or clearance, the list of possible fault locations (*mylist*) is searched for given search length (*fault_length*) and fault location, (*fault_loc*), fed back by the FPM (Fig. 4). For all search lengths, the search space is translated in the available bit space by N simulation clock cycles (with $N-1$ wait cycles, followed by an iteration cycle), which is given by

$$\begin{aligned} N &= 1;\ search\_length \leq list\_size\_in\_bits \\ &= \lceil \frac{search\_length}{list\_size\_in\_bits} \rceil;\ otherwise. \end{aligned}$$

This gives one list iteration within the fault locations' database per clock cycle for single fault injection or two list iterations for two faults' injection and so on. The registers in the list within the database, *mylist*, are converted

to their original types and size within the FIM and different types of faults are implemented with appropriate operations replaced in bold commented place in *insert_faults* (Fig. 4). A list of possible faults (temporary and permanent) and necessary parameters and actions needed are shown in Table 3.3. For simplicity, symbol for each parameter is used: L meaning *search_length*, r meaning *fault_loc*, τ meaning *fault_duration* and r' meaning *approx_fault_loc*.

Fault types	Fault injection technique
Stuck-at-fault	A bit at r in L is changed to '1' or '0' for stuck-at-1 or stuck-at-0 faults for τ cycles (if temporary) or indefinitely (if permanent)
Bit-flip/SEU	A bit at r in L is XORed with '1' for one clock cycle duration
Delay	Original SystemC *wait()* statements are replaced by *wait(N)* statements, where N is registered in the fault database. A bit close to r' in L is bit flipped for such types
Indetermination	A logic type value close to r' in L is changed from 'Z' to 'X' for τ cycles (if temporary) or indefinitely (if permanent)

Table 2. Fault types and injection techniques

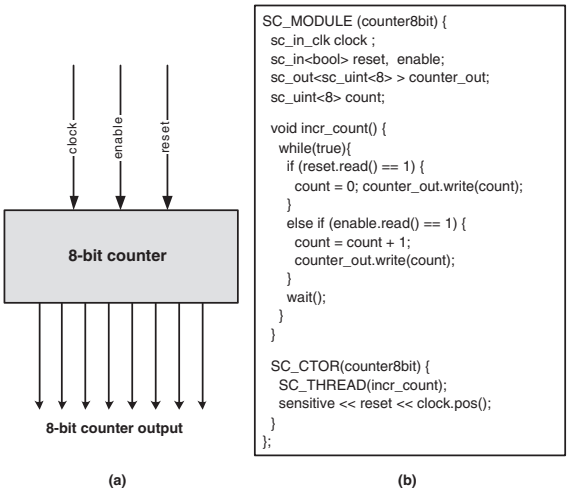

Figure 5. (a) Block diagram of a sync. 8-bit counter, and (b) SystemC description of a sync. 8-bit counter

4 Motivations and Comparisons

In this section, we demonstrate the capabilities of the proposed FI technique and compare it with saboteur, mutant and simulation command-based approach (such as [3]). Using the motivational SystemC example setup, the following comparisons are carried out.

Comparison 1: Simplicity and Intrusiveness

Inclusion of saboteurs or mutating behavioural descriptions in saboteur or mutant-based approaches require direct intrusion into the description and are not suitable for higher- or mixed-level systems (Sec. 2.1). The proposed technique employs powerful simulation command-based approach and can be applied to a design with minimum intrusion, similar to [3]. To demonstrate and facilitate comparison of simplicity and intrusion, first a block diagram of a synchronous 8-bit counter and a SystemC description of the same is presented in Fig. 5(a) and Fig. 5(b). SystemC descriptions using [3] and our proposed technique are presented in Fig. 6(a) and Fig. 6(b).

Using observational comparison among the SystemC descriptions in Fig. 5 and 6, it can be seen that the proposed technique (Fig. 6(b)) requires replacement of original types *bool* and *sc_uint<8>* (Fig. 5(b)) to their equivalent FI enabler types (from Table 3.1) *Reg<bool>* and *UIntReg<8>* (marked block 1 in Fig. 5(b)), requiring inclusion of proposed type definitions' header file *FIReg.h* (marked block 2 in Fig. 5(b)). The simulation command-based approach [3] requires these types to be converted to *sc_logic* and *sc_lv<8>* (Fig. 6(a)) and using the extra types (Sec. 2.2) to inject faults from testbench or top-level module. Such conversion require extra programming effort as shown in marked block 2 in Fig. 6(a) to keep the functionalities similar.

Comparison 2: Fault Representation and Capabilities

Simulated FI approaches rely on perturbation of registers. These approaches are based on the fact that faults at the hardware level are propagated towards registers at higher abstraction levels (in data or signal registers), often referred to *multiplicity*. Although the effect of faults in the inaccessible registers is not negligible but it scales much in the similar way as the architected higher level registers that appear in HDL descriptions [9]. A desirable feature of a simulated FI technique is to have access to all visible data and signal registers of a model description for better fault access and representation [1]. Saboteurs and mutants, as described in Sec. 2.1, have poor access to variable manipulation and requires expensive re-design using manual or behavioural mutation to introduce fault injection. Simulation command-based approach [3] restricts types to *sc_logic* and *sc_lv<N>* only and has disadvantages as follows. *Firstly*, he value space in [3] increases to seven values from four (Section 2.2), with default value to 'X'. To avoid undesired initialisations in SystemC *sc_start(..)* needs to be replaced by *sc_initialize* and *sc_cycle* (to control simulation cycles). Our proposed technique keeps the value space unchanged by simple replacement of types (Table 3.1). *Secondly*, original types, such as *int[], float[] etc.* and *sc_logic* or *sc_lv<N>* are not always directly compatible with each other and often require manual code modifications and intrusions (such as, marked block 2 in Fig. 6(c)).

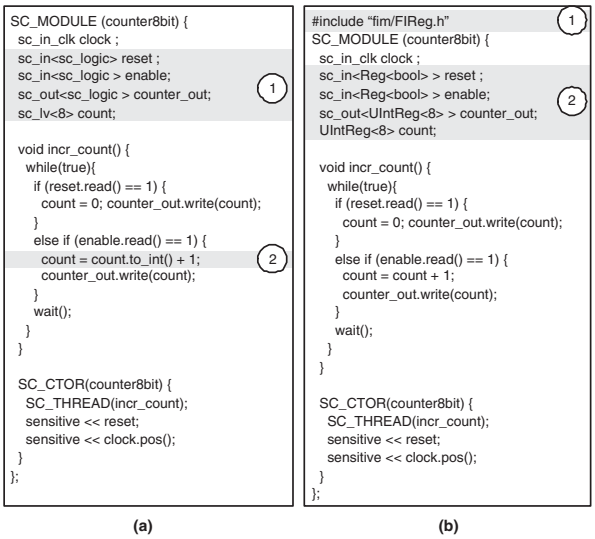

Figure 6. SystemC descriptions of (a) a Sync. 8-bit counter using [3], and (b) a Sync. 8-bit counter using proposed technique

The proposed FI technique can simulate major fault types, including SEUs, stuck-up faults, (Sec. 3.3). It can carry out *directed* or *non-directed* fault injection. For direct fault injection into a target signal and data register, first the location of the register in the list is obtained by

```
long loc = FIMgr::getInstance().getLocation
(DUT.target_register.get_ptr());
```

and fault location is set manually by

```
FIMgr::getInstance().setFaultLocation(loc);.
```

For *non-directed* fault injection, the member functions of FPM, *setProbability(..)* and *setProbabilityType(..)* are used within the testbench as described in Sec. 3.2 to access different fault types and their associated probabilities from testbench. Similar *non-directed* SystemC-based FI technique using saboteurs is reported in [4]. Fault injection technique in [3] can employ direct method only.

Comparison 3: Speed of Simulation

Speed of simulation is often used as a benchmark for simulator performance [9]. A D-type flip-flop (DFF) and a synchronous 8-bit counter were designed and simulated along with their testbenches using saboteur, mutant, simulation command-based approach [3] and the proposed technique for fault injection. The SystemC simulation times recorded as average elapsed time on RHEL 2.6.9-42 with Intel Pentium-4 CPU at 3.20GHz clock speed and 1GB RAM are shown in Fig. 7 for the following four cases: *i)* DFF for 5 SEUs/10000 clock cycles, *ii)* 8-bit counter for 5 SEUs/10000 clock cycles, *iii)* DFF for 5 stuck-at-0 faults/10000 clock cycles, and *iv)* 8-bit counter for 5 stuck-at-0 faults/10000 clock cycles. As shown in Fig. 7, the proposed technique has less simulation times than saboteurs and mutant based FI techniques by on average 12.76% and 25.42%, respectively for four test cases. The proposed technique and simulation command-based approach [3] have comparable simulation times with only 1.75% higher simulation time on average due to dynamic list building in possible fault locations' database.

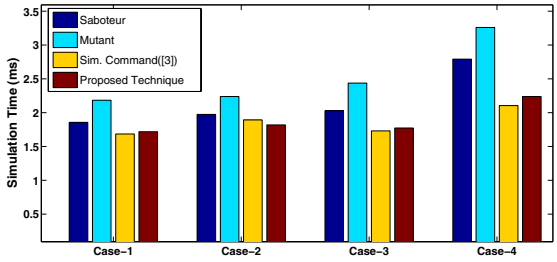

Figure 7. Comparison of simulation speed for fault simulation techniques

5 MPEG-2 Decoder

In this section, we set up an MPEG-2 decoder to validate fault injection (SEUs only) abilities using the proposed technique (Fig. 1). Due to type restrictions, such setup is not feasible by [3] (Sec. 2.2). To the best of our knowledge, no such setup using SystemC-based FI methods have been reported previously.

A cycle-accurate behavioural SystemC model for MPEG-2 video decoder was developed following [11]. To enable fault injections, all primitive C++ or SystemC data and signal types were replaced by FI enabler types (Table 3.1). Due to automatic registration and de-registration (Sec. 3.1), the list size (i.e. the sum of all register sizes within the database of possible fault locations) varies and is averaged over total simulation time of 41928 clock cycles as 49323 bits. Constants within pre-processor directives cannot be included by the proposed technique, which sum up as a approximate total of extra 568 bits. As such, up to 98.9% of visible and on-the-fly registers can be subjected to fault injection within MPEG-2 video decoder, giving it very high fault access and representation.

Different simulations were carried out with varied fault probabilities with uniform distribution within the bitspace for injecting SEUs within registers and memories of SystemC MPEG-2 decoder description. The simulations were recorded on Intel Pentium-4 CPU clocked at 3.20GHz system with 1GB RAM running SystemC on RHEL 2.6.9-42 to

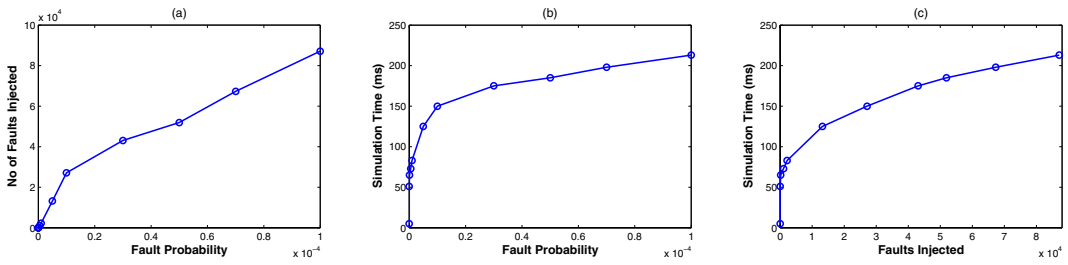

Figure 8. (a) Fault probability against total faults injected, (b) Fault probability against simulation time (ms), and (c) Faults injected against simulation time

decode a video with 3 frames and 93 encoded macroblocks with picture size of 252×288 pixels. The recorded total SEUs injected and simulation times (as CPU elapsed time) with varied fault probability are shown in Fig. 8(a) and Fig. 8(b). The total injected SEUs with corresponding simulation times is shown in Fig. 8(c). No more than (1×10^{-4}) could be carried out due to early termination of application caused by faults injected.

It can be seen from Fig. 8(a) that the total SEUs injected varies almost proportionately with fault probability due to conversion of fault probability in bitspace into timespace (Sec. 3.3). However, the simulation times increase sharply (from 5ms to 50ms) when injected faults increase from 0 to 201 corresponding to fault probability increase from 0 to 5×10^{-7} (Fig. 8(a) and (b)). As the fault probability increases, the simulation times gradually saturate due to faults injection on almost every clock cycle (Fig. 8(b)). Referring to Fig. 8(c), a total of 52371 and 87682 SEUs were injected over 41928 clock cycles with corresponding simulation times of 181ms and 217ms, respectively. The increase in the simulation times from fault-free case to faulty cases are generally expected and in this case the search times of fault locations in the list are also included. The simulations times are still low (in the order of *ms*) with the given advantage of 98.9% fault representation, minimum intrusive design by simple change of data types (Table. 3.1).

6 Conclusions

We have presented a new SystemC-based FI technique with improved fault representation using replacement of the original C++/SystemC types by proposed FI enabler types. In doing so the the functions are kept intact and design modifications are kept simple and less intrusive. Using example circuits, we have shown that the proposed technique has less or comparable simulation times as recently reported techniques. An MPEG-2 decoder based experimental setup is also presented to demonstrate the fault injection capabilities of the proposed technique in a high-level behavioural system. We have shown that our technique has 98.9% fault representation, acceptable speed of simulation and requires less modification to enable fault injection.

Acknowledgment

The authors would like to thank the EPSRC (UK) for funding this research in part under grants EP/C512804/1 and EP/035965/1.

References

[1] J. Gracia, J. Baraza, D. Gil, and P. Gil, "Comparison and Application of Different VHDL-based Fault Injection Techniques," in *Proceedings of the IEEE DFT'01*, San Francisco, CA, USA, 2001, pp. 233–241.

[2] S. Misera, H. T. Vierhaus, L. Breitenfeld, and A. Sieber, "A Mixed Language Fault Simulation of VHDL and SystemC," in *Proceedings of DSD*, 2006, pp. 275–279.

[3] S. Misera, H. Vierhaus, and A. Sieber, "Fault Injection Techniques and Their Accelerated Simulation in SystemC," in *Proceedings of DSD*, Aug. 2007, pp. 587–595.

[4] A. Fin and F. Fummi, *Languages for System Specification*. Springer US, 2004, ch. LAERTE++: An Object Oriented High-Level TPG for SystemC Designs.

[5] K. Chang and Y. Chen, "System-level Fault Injection in Systemc Design Platform," in *Proceedings of 8th International Symposium on Advanced Intelligent Systems (ISIS)*, 2007.

[6] K. Rothbart, U. Neffe, C. Steger, R. Weiss, E. Rieger, and A. Muehlberger, "High Level Fault Injection for Attack Simulation in Smart Cards," in *Proceedings of the ATS'04*, Nov. 2004, pp. 118–121.

[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," in *Proceedings of IEEE ASPDAC*, 2008, pp. 673–678.

[8] A. Castelnuovot, A. Fin, F. Fummi, and F. Sforza, "Emulation-based Design Errors Identification," in *Proceedings of the IEEE DFT'02*, 2002, pp. 365–371.

[9] P. Gil, H. Madeira, and J. Arlat, "Fault Representativeness," LAAS-CNRS (France), Tech. Rep., 2002, project funded by the European Community under the IST Programme (1998-2002).

[10] J. Baraza, G. J., D. Gil, and P. Gil, "Improvement of Fault Injection Techniques based on VHDL Code Modification," in *Proceedings of Tenth IEEE International High-Level Design Validation and Test Workshop*, 2005, pp. 19–26.

[11] ISO, "ISO/IEC 13818-2: 1995 (E)," http://www.iso.org.