Testbench Components Verification Using Fault Injection Techniques

N. A. Banciu[1], G. Toacse[2]

[1] Siemens Program and System Engineering Braşov, Corporate Technology Central Eastern Europe Department, Romania, nicolae.banciu@siemens.com

[2]Transilvania University of Brasov, toacseg@vega.unitbv.ro

Abstract- **New methodologies for digital designs verification make use of SystemVerilog's object-oriented mechanisms to speed-up the verification environment development. Yet, undetected testbench errors can slow down or even compromise the overall verification process. This paper concentrates on fault injection technique applied to different SystemVerilog testbench components. By altering functionality in different places of the testbench, potential hidden errors can be detected, improving the testbench capacity to detect design misbehavior. The fault injection in SystemVerilog testbench components may be used in addition to existing methods of functional verification analysis, for testbench validation. Modalities to alter the main testbench components are presented, highlighting the effects on the testbench behavior.**

I. Introduction

Functional verification is used in digital circuits design to check the implemented behavior is correct and follows exactly the predefined specifications. To keep up the designs complexity grow, the functional verification faces continuous changes and improvements. Thus, dedicated languages and methodologies were created to address the verification's particular needs. Commonly known as Hardware Verification Languages (HVL) languages like PSL (Property Specification Language), *e*, SystemVerilog [1], OpenVera or SystemC have constructs that check design behavior in different ways. Based on these languages, a broad range of verification techniques were developed. Techniques like formal analysis, assertion-based verification, constraint-random stimulus generation or coverage-driven verification [2] are constantly used by verification engineers for checking and validating designs functionality. Each of these techniques has its own strong and week points and, for better results, they are often used in a combined way. Complex verification environments are created to benefit the power of assertions while applying constraints for random generated parameters in the same time. Such environments allow better verification of complex designs but the time spent for their development is significantly higher. The complexity involves also a larger number of errors introduced in the verification environment, endangering the verification process if not detected.

Using methodologies like VMM Verification Methodology (VMM) [3], Open Verification Methodology (OVM) [4] or *e* Reuse Methodology (*e*RM™) [5] the development time is reduced and the verification components can be reused in other projects. These methodologies benefit the flexibility of object-oriented (or aspect-oriented [6]) mechanisms offered by languages like SystemVerilog, OpenVera and *e*. The verification environment - as described by these methodologies - comprises of a testbench surrounding the Device Under Test (DUT). Customizable predefined elements are offered to create the main testbench components.

Either reusing existing parts or creating everything from scratch, the testbench can introduce its own functionality errors. Interfaces misinterpretation, incorrect start and stop conditions of certain iterations, different corner cases treated improperly, erroneous state-machines, or incorrect Object-Oriented (OO) constructs are just some examples of the errors usually introduced by testbench developers. These errors are observed later, at simulation time. Whenever a difference is found between expected results and actual DUT behavior, a close analysis is performed to see if the problem resides in DUT or in testbench, due to implementation or misinterpretation. Such an investigation may take time and additional effort, increasing the cost and overall verification time. Moreover, even if simulation reports no errors, there is still need to know if the testbench is reliable or not and all its constructs are active during simulation.

The present paper concentrates on the modality to verify the testbench components described in SystemVerilog, by altering the components constructs. Introducing artificially created errors in testbench components may reveal testbench hidden errors or week points which decrease the capacity to detect design errors. The approach of applying fault-injection technique to the SystemVerilog constructs used in testbenches was also treated in [7] (in press).

The paper is structured as follows. Section II presents an example of class-based SystemVerilog testbench architecture. Section III presents the presence of errors in testbench components and the existing methods for verification process analysis. Section IV shows modalities to alter some of the SystemVerilog's OO constructs and the main testbench components, together with the expected simulation results while the conclusions are drawn in Section V.

II. Class-Based Testbench Architecture

Traditional testbench approaches use VHDL or Verilog as development languages. Their limitations in terms of constructs for functional verification imposed creation of more flexible languages. Created as an extension of IEEE 1364-2001 Verilog Hardware Description Language (HDL),

978-1-4244-7020-4/10/$26.00 ©2010 IEEE

SystemVerilog provides support for all three main types of verification: directed and constrained-random stimulus generation, coverage-driven verification and assertion based verification [2]. Being rapidly adopted by both EDA vendors and verification engineers, SystemVerilog is becoming one of the most used verification language. Using OO mechanisms which allow abstraction and encapsulation, SystemVerilog leverages fast testbench creation and testbench reuse. Its OO features allowed development of new verification methodologies (like VMM and OVM) consisting of testbench architecture definition and a set of rules for creating new components or customizing and using the existing ones. Both methodologies define architectures and different components that can be used for injecting stimuli and observing DUT responses and reactions. In this article we use a SystemVerilog testbench with components as they are defined in VMM methodology, described in [2]:

- *Testcase*: controls the behavior of a simulation. Each testcase can perform direct actions or can apply constraints to explicitly conduct the verification to desired corner cases.
- *Generator*: creates stimuli (or reads stimuli from files) as sequences of transactions and sends the transactions to driver. Each transaction generated is transmitted in parallel to Scoreboards.
- *Driver*: translates incoming transactions into signals. Has knowledge of in-out protocol and stimulates the DUT through interfaces, at the I/O ports.
- *Monitor*: collects DUT responses and send them to protocol checkers and scoreboards.
- *Checker*: analyzes DUT responses according to predefined protocol rules.
- *Scoreboard/Ref. Model*: calculate expected results or implement lists of transactions with check mechanisms.
- *Interface*: specifies the communication signals between DUT and verification components. Implements the clock skews used to drive or sample these signals, allowing a cycle-based communication with DUT.

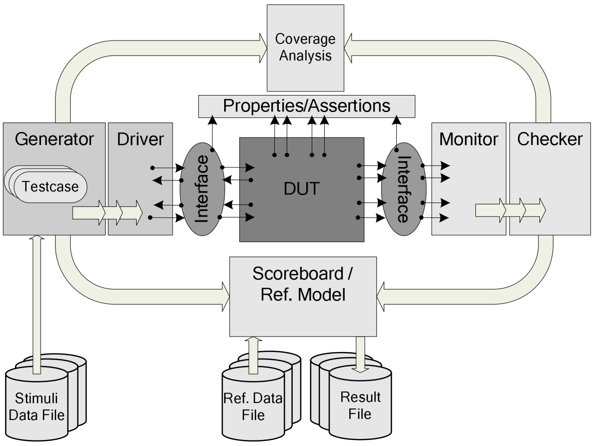

Fig. 1. Example of testbench architecture

Additional testbench elements may be present. Thus, if the data to be injected (stimuli) is generated externally or external reference models are used, stimuli data files or expected results have to be read into. For this, *File I/O* testbench elements must be used, either as independent testbench components or included into generators or scoreboards. A simplified SystemVerilog-based testbench architecture, with components and interconnections based on the VMM architecture described in [2] is depicted in Fig. 1.

Existing verification methodologies provide a set of rules for testbench components and a collection of predefined classes to support these rules. Creating and interconnecting the testbench components by extending these classes is faster and safer in comparison to the approach of creating them from scratch. Provided base classes can be extended to create all the main components (stimuli generators, drivers, monitors, scoreboards, checkers) and the surrounding environment. The user must customize them to implement the testbench that best fits the verification needs.

III. Presence of Errors In Testbench Components

Creating an ad-hoc class-based testbench or using a methodology with predefined classes, testbench development is not an error-free process. Verification engineers often introduce errors, either caused by misunderstanding of the specifications or by accident. Some of these errors will be detected at compile, elaboration (when all components are bind and the design is prepared for simulation [1]) or run-time. Some others (especially those related to testbench components behavior) may remain undetected during the entire verification process. Starting with the first compile phase, verification engineers perform a multi-iteration task to remove the bugs residing in the testbench code. When all syntax errors are corrected, the DUT and surrounding testbench are passing the elaboration phase. Incompatibilities between testbench components or incorrect DUT interfacing are reported in this phase. At simulation, incorrect usage of different dynamic elements or accesses to non-existent objects results in run-time error. This is the last check performed automatically by the simulator. Once this barrier is passed, it is up to the implementation to report or skip other potential errors. Usually the whole effort is spent to check DUT functionality and to assure its potential errors are made observable to the testbench. Only in some cases (e.g. in testbench debug process) effort is being spent to increase the observable level of testbench potential errors.

SystemVerilog's OO mechanisms (like inheritance and polymorphism) applied to user-defined data types (classes) allow abstraction and encapsulation. The mentioned verification methodologies implement their base classes in respect to the abstraction and encapsulation principles. Still, is up to the user to maintain these principles in its derived classes. If these principles are violated the testbench may end-up with errors deep inside the code, difficult to be detected. As in all OO implementations, the errors may reside [8][9] either inside the functions, between functions members of the same class or between the classes. In many cases, incomplete or incorrect implemented behavior is never reported. Erroneous DUT responses might pass the checkers without any notifications. Incorrect DUT states may not trigger any action when checked against reference model or invalid transactions may be skipped by scoreboards. These are the most dangerous errors as they reduce the testbench detection capability. The other errors, reported by the simulator, even are the most destructive making the testbench unusable, are easy observable.

As described also in [7], the errors affecting compile or elaboration process can be classified as *static errors* while those which occur during simulation time (directly observable or not), *dynamic errors*. From the implementation point of view, the errors can be classified as language-specific and methodology-specific errors. The language-specific errors are strictly related to the language constructs, e.g. syntax errors, incorrect statements, data type errors, incorrect assignments, invalid constraints or assertions (examined in detail in [7]). Methodology specific errors are related to the implementation and usage of testbench components as part of a particular methodology (like VMM or OVM). Such errors reside in the components functionality or the way they are interconnected, activated and deactivated.

Intuitively, in the testbench debug phase, verification engineers deliberately change the functionality of few testbench elements and then start the simulation to see the results. If the error introduced changes the simulation results as expected, the respective testbench element is correctly implemented. If not, a detailed analysis is performed to determine the presence of implementation errors. But this approach is done only when they suspect a hidden error resides in the code. Most of the time they rely on the other existing methods for verification process analysis, like code coverage [10] or functional coverage [11]. Yet these methods cannot fully predict the presence of hidden errors in the testbench that might lead to incomplete or incorrect verification. Code coverage checks that every line of interest (statements, expressions, branches) [12][13] within the DUT code was exercised during simulations of all defined tests but reports nothing about interactivity. Functional coverage is based on coverage statements introduced in the testbench code based on functionality features extracted from the Verification Plan. But incomplete list of features to be checked or incorrect analysis of DUT behavior let hidden bugs undetected.

Other approaches use fault injection techniques to deliberately alter parts of the DUT code. The fault injection technique is not new. It has been adopted from software testing (mutation analysis) [14][15] and has been successfully adapted for digital designs. There are three main categories of fault injection techniques [16]: at physical level (Hardware Implemented Fault Injections- HWIFI) by affecting the physical circuit pins logic value, at software level (SWIFI) by introducing in software the errors that might occur in hardware and simulation-based, by altering the logical values during simulation. The fault-injection is used in digital design by altering the VHDL or Verilog code. Mutation is based on constructs like *saboteurs* (components added to the original code to alter the value or the timing characteristics of a signal) or *mutants* (components that replace another component) [16][17]. Using saboteurs or mutants, the simulated circuit's fault-tolerance mechanisms can be analyzed or, for testing purposes, the number and size of test patterns can be reduced [16]. Several automatic tools have been developed to insert artificially created bugs in VHDL models [16][18][19]. Other approaches are creating mutations at system level directly in SystemC TLM primitives while elaborated commercial tools were created to analyze the verification environment by checking the testbench capability to detect different errors injected in the design [20].

The present paper focuses on the approach of applying the fault injection techniques to the testbench components described in SystemVerilog. After the testbench is considered to be fully functional, the possible mutations to be applied are analyzed and injection of all identified possible faults can be performed. The injection of methodology-based errors can be completed with injection of different language-specific errors (approach treated in [7]). Batch mode simulation results can be automatically checked by comparison of result files in both cases, with and without mutation. The results between simulation with and without injected errors must be different to reflect the correct testbench functionality. If the injected errors have no effect, hidden problems may reside in the testbench.

IV. Fault Injection In Testbench Components

A. Mutation Techniques Applied to SystemVerilog Object-Oriented Constructs

Class-based SystemVerilog testbench implementations can use mutations related to OO features. Thus, class data type mutations [21] can be created to affect the way data classes are declared or referred, the inheritance, data encapsulation mechanisms and polymorphism.

Already used in software testing, the main OO features (access control, inheritance, polymorphism and overloading) have specific mutations defined [9][22]. Within a class hierarchy, where classes are inherited one from each other, mutations can be achieved by changing the type of an object with another one to check polymorphism. From case to case, such mutation may result in compile errors (or runtime errors) as the new type may not (or differently) implement the intended behavior. An example is shown in Fig. 2 a) and b). Polymorphism may also be checked by changing the run-time type of an object (with one from the class inheritance hierarchy). A different constructor will be called and the object will have characteristics of the new type as shown in Fig. 2 c). Another common mutation used for polymorphism analysis is related to object initialization [22]. Incorrect initialization is a frequent error and may be checked by altering the constructor's parameters list, objects being created with different initial values or with a different type. Usage of these mutations and careful analysis of the results lead to detection of unintended errors in objects definition and instantiation. If no visible effects are produced by applying mutations, additional constructs meant to distinguish between possible types should be added in the class definitions and the mutated code re-run.

```
class short_packet extends packet;
class long_packet  extends short_packet;

short_packet tr_data=new();

//mutation a)
packet tr_data=new();
short_packet sh_pkt_obj=new();
tr_data = sh_pkt_obj;

//mutation b)
long_packet tr_data=new();
short_packet sh_pkt_obj=new();
tr_data = sh_pkt_obj;

//mutation c)
packet pkt_data_obj=new();
tr_data = pkt_data_obj;

//mutation d)
long_packet lng_data_obj=new();
tr_data = lng_data_obj;
```

Fig. 2. Possible mutations to check polymorphism

SystemVerilog allows definitions of virtual methods in the base class (the superclass) which may be overridden in the derived classes (subclasses). In subclasses, the overridden method implements a different functionality than in the supeclass. To check correct usage of these functions, several mutations can be applied. Removing the definition of virtual methods in derived classes is one of them. This may lead to references to the original overridden method (from superclass) instead of the intended (from subclass). Changing the name of overridden methods, changing the way they are implemented or the position within the code they are called should also trigger different behavior.

To check correct encapsulation, mutations can be performed to change the access mode of class members. By default, all SystemVerilog class members are public and can be used in any place the object of the respective class is present [1]. If the access modifier `local` is used, the respective property or method cannot be accessed outside the class. To make it available to derived classes, the `protected` modifier must be used instead. Mutations applied here (e.g. by changing `local` to `protected` or other possible combinations) help indentifying possible encapsulation problems. In order to share a property value for all instances of a class (objects), the `static` qualifier is used. Used for methods, the `static` qualifier affects either the method's lifetime inside the class (if the `static` qualifier precedes the function's name) or its arguments and variables lifetime (if the qualifier follows the function's name). Mutations based on changing the position of this qualifier (in case of methods) or removing it completely (applicable for both methods and properties) are useful to identify incorrect lifetime definitions inside the code. To access properties and methods of superclass, the subclass must use the `super` keyword. If the keyword is omitted, the properties from the subclass are used instead. In case of multiple instances of the same class, using the keyword `this` identifies the members of current object instance [1]. Removing it may also identify other hidden errors in class functionality, as shown in Fig. 3.

```
class transmit_data extends eth;

  integer len = 512; //default length

  function new (integer len)
    this.len = len;
    ...
  endfunction
  ...
endclass
```

Fig. 3. Removing keyword *this* will not change the default length

A. Fault Injection in Testbench Components

Different mutations can be applied to the main testbench components to alter their functionality. The most important mutations are to be applied to stimuli generators (affecting the way the DUT is stimulated) or to response checkers (affecting the way that testbench analyzes DUT responses and reports the errors). The mutated code is then re-simulated observing the new behavior.

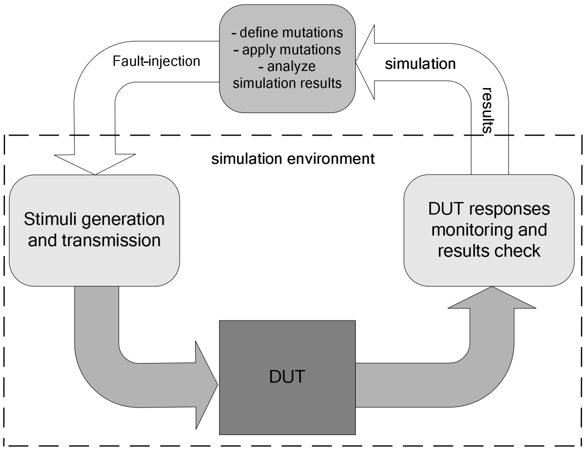
- define mutations
- apply mutations
- analyze simulation results
Fault-injection
simulation
results
simulation environment
Stimuli generation and transmission
DUT responses monitoring and results check
DUT

Fig. 4. Injecting faults into the testbench components

If identical results are obtained, the error injected in the testbench code is either not observable or the implementation is wrong and a careful analysis must be performed for possible testbench errors. Yet, not all mutations are capable of changing the simulation results, in some cases the injected error being harmless to the testbench (either the mutation was in the acceptable limits or the testbench has a certain limit of fault tolerance). Several possible mutations on the main testbench components already mentioned in Section II (as in VMM methodology, described in [2]) are presented in the following sub-sections.

1) Altering Generators

In constraint-random verifications, generators produce random sequences of transactions (stimuli data) and send them to drivers to be injected into DUT. Generators may define, using constraints, the number and the type of sequences to be generated within a simulation run. Transactions are defined as data classes with different properties and the generators make use of these properties to create different scenarios. The random generation functionality may be altered with a few possible errors injected in its parameters. Thus, mutations can be applied to the constraints defining the number and the type of transactions to be created. Altering the number of transactions allows observing the generation behavior. This could have impact on the simulation duration and could be used to identify incomplete stimulation sequences. Setting the generators to continuously generate transactions can be used to check the simulation is not blocked in long simulations. To alter individual transactions, the parameters that configure the generated transactions type must be identified within the code and constraint to different values. This should result in a change of data format being transmitted and must trigger different DUT behavior. In the example shown in Fig. 5, the scenario generator that creates the `data_eth` transactions contains a constraint for the data length. Mutation applied will result in shorter packages being generated and transmitted to DUT through the driver. The shorter packages must determine a different DUT behavior (e.g. can be dropped if they have an illegal length).

```
class test_scenario extends data_eth_scenario;
   ...
   constraint short_package {
      length == 512;
      repeated == 0;  }
   ...
endclass

class test_scenario_mut extends data_eth_scenario;
   ...
   constraint short_package {
      length == 510;
      repeated == 0;  }
   ...
endclass
```

Fig. 5. Altering the scenario generator constraints

Generators behavior and communication with the drivers is implemented using different methods. The mutations can be applied at class level, at constraint blocks level or at methods level. Altering the methods behavior to trigger different aspects of interest in data generation flow extends the generators analysis. Functions with mutated behavior or different bus dimensions used in communication with drivers may disturb the way the stimuli are injected to the DUT. The results should show significant change at simulation run or run-time errors. If data is not generated locally but read from external files (e.g. when stimuli are created with external applications), the generators must contain functions for files access. They are subject for additional mutations. Changing the names of the files or removing the statements where files are opened must result in run-time errors. If not, the respective files are never used so incorrect data is being sent to the driver to be injected into the DUT.

2) Drivers Mutations

Drivers receive data transactions from generators and inject them to the DUT through interfaces. Similar to generators, drivers contain different functions, used to implement the communication protocol with DUT. At this level, the DUT signals are used (through interfaces) for synchronizations and data injection. The most important mutations are to be created in the functions that implement the transmission protocol. Synchronization on the opposite clock edge, clock cycles insertion or removal, change of flow (e.g. by moving r/w operations to different points) are some of the potential mutations which can accurately characterize the driver‘s functionality. Any modification in the protocol rules affects the communication with the DUT and should compromise the stimuli injection. If the mutations produce no simulation errors, the DUT could be incorrectly stimulated. An example is shown in Fig. 6 where data is sent 4 clock cycles after request was received. Mutation applied will send data earlier than DUT expects it.

```
task driver::drive_data();
   ...
   @(this.dut_in_intf.cko);

   forever begin
      while(!this.dut_in_intf.cko.req)begin
         @(this.dut_in_intf.cko); end

      for (int i=0; i<4; i++) begin
         @(this.dut_in_intf.cko); end

      dut_in_intf.cko.data <= data_tr.data;
   end
endtask: drive_data


task driver::drive_data_mut();
   ...
   forever begin
      ...
      for (int i=0; i<2; i++) begin
         @(this.dut_in_intf.cko); end

      dut_in_intf.cko.data <= data_tr.data;
    end
endtask: drive_data_mut
```

Fig. 6. Mutation on driver synchronization

3) Altering Monitors and Checkers

At the DUT outputs, the monitors collect signal-level data through interfaces and send it for further analysis. Checking of the receive protocol may be implemented in separate components or inside the monitor itself, using different tasks or functions. After protocol check, DUT sampled signals are packed into data objects (transactions) and sent for analysis against reference models or to scoreboards. Similar to drivers verification, the synchronization with the DUT can be altered by injecting errors in the signals sampling logic or in the logic responsible for the protocol check. This should give an overview of the monitoring process and the implemented protocol check. Applying mutations on the protocol check should generate simulation errors if the check was correctly implemented.

4) Mutations in Scoreboards

After protocol rules check, data transactions are compared against data from reference models inside the scoreboards. The reference models can be either part of the testbench (supplying reference data to scoreboards) or may reside in external applications (reference data files being used in this case). When using external data files, data read from files is packed inside the scoreboard into data objects (the same format as the data received from monitors). Reference data received from the reference model (or from external files) is compared with data from monitor using dedicated methods. The most effective way to analyze the correctness of the comparison is to create mutations in these methods. Such an example is shown in Fig. 7 where errors will be reported instead of successful match. If the reference model is implemented inside the testbench, the mutations can be extended to alter its behavior. Changing the way the reference model calculates the expected data must trigger immediate comparisson errors.

```
virtual function rcv_data::compare(data_t to,
                     output int diff);

     rcv_data received_d;
     ...
     if (this.data !== received_d.data) begin
         diff = (this.data - received_d.data);
         err = 1;
     end
     ...
endfunction: compare

virtual function rcv_data::compare_mut(data_t to,
                     output int diff);
     ...
     if (this.data === received_d.data) begin
         diff = (this.data - received_d.data);
         err = 1;
     end
     ...
endfunction: compare_mut
```

Fig. 7. Affecting the receive data comparrison

Scoreboards may contain dynamic lists (also known as queues) to check data transactions. Every time a transaction is injected into the DUT, the respective transaction is added to a list. The transactions received from DUT are then searched inside the list and deleted. If a transaction is not found in the list, an error is generated. Also, at the end of simulation the list must be found empty otherwise some of the transactions were not processed by the DUT. Predefined functions [1] are available for lists data manipulation (e.g. `pop_front()`, `pop_back()`, `push_front()`, or `push_back()`) to insert or extract elements to/from the list. Mutations can be introduced by changing the place of insertion or extraction (e.g. swapping `pop_front()` with `pop_back()`) thus mixing the data flow. Also, any access to a non-existent location of a list (outside the list borders) will generate a run-time error. If the mutations have no effect, the logic behind scoreboard implementation must be re-analyzed.

5) Altering the Verification Environment

All the main components are instantiated within the verification environment. There, they must be connected and activated. Removing a component instance or eliminating it from the activation list are simple mutations and are often performed by testbench developers in the early phases of testbench debug. Components which are not exercised during simulation can be immediately observed. In other cases, delaying the activation moment of a component (e.g. activating it at a specific moment, from the test), allow observations related to inter-components synchronizations (e.g. synchronization between monitors and scoreboards or between drivers and generators). In complex testbenches, with multiple components (e.g. with multiple monitors and scoreboards) a larger number of components interconnections is needed. But connecting each pair of components can be error-prone, especially for identical components (multiple instances of the same component). Using simple mutations to interchange the connections between components instances, these problems can be easily identified.

V. Conclusions

A small number of the possible mutations applied to testbench components or its OO constructs have been exemplified to show how fault-injection techniques may be used in a simple way to analyze the verification environments. The analysis may be extended to other constructs or testbench components. Used systematic, together with language-specific errors injected in the testbench, a broad range of hidden problems can be identified, shortening the time for testbench debug. The approach of fault-injections in testbench can be used also as a complementary method for testbench validation. Using automated software tools or even simple ad-hoc created scripts, the testbench code can be analyzed to identify possible mutations to be applied. Simulations run and results analysis can be performed also automatically, the effort of automation being fully rewarded by reduced testbench development time and higher testbench reliability.

The SystemVerilog Language Reference Manual (LRM) was specified by the Accellera SystemVerilog committee as an extension of Verilog 1364-2001. SystemVerilog language is IEEE 1800-2005 standard, VHDL language is IEEE 1076 standard, Verilog language is IEEE 1364 standard, *e* language is IEEE 1647 standard and SystemC® language is IEEE 1666 standard. Verilog is a registered trademark of Cadence Design Systems, Inc. OpenVera™ language is property of Synopsys, Inc. VMM methodology was created by Synopsys, Inc and implementation was later donated to Accellera. OVM was created by Cadence Design Services, Inc. and Mentor Graphics, Corp. Other terms and product names may be the trademarks of others.

References

[1] *SystemVerilog 3.1,Accellera's Extensions to Verilog®*, Accelera 2003, http://www.eda.org/sv/

[2] J. Bergeron, E. Cerny, A. Hunter, A. Nightingale, *Verification Methodology Manual for SystemVerilog*, 1st ed., Springer 2005

[3] *VMM Methodology*, http://www.vmmcentral.org/

[4] *OVM Methodology*, http://www.ovmworld.org/

[5] *e Reuse Methodology*, Verisity Design, Inc., http://www.verisity.com/products/erm.html

[6] Y. Hollander, M. Morley and A. Noy, “The ***e*** Language: A Fresh Separation of Concerns,” in *Proc. of the Tech. of Object-Oriented Lang. and Syst.*, pp 41, March 2001

[7] N.A. Banciu and G. Toacse, “Fault Injection Technique Approach for Testbench Analysis,” in press for *2010 IEEE Intl. Conf. on Aut., Qual. and Test., Robo. AQTR 2010.*

[8] M. J. Harrold and J. D. McGregor, “Incremental testing of object-oriented class structures,” in *Proc. of 14th Intl. Conf. on Softw. Eng.*, pp. 68-80, Melbourne, Australia, 1992.

[9] Y.-S. Ma, J. Offutt, and Y. R. Kwon, “Mujava: an automated class mutation system,” *Softw. Testing, Verif. and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[10] S.V. Kodakara, D.A. Mathaikutty, A. Dingankar, S.Shukla and D.J. Lilja, “A Probabilistic Analysis For Fault Detectability of Code Coverage Metrics,” *IEEE Micro. Test Verifi. (MTV) Work-shop*, Nov. 2006.

[11] S. Tasiran, K. Keutzer, “Coverage Metrics for Functional Validation of Hardware Designs”, *IEEE Des. Test Comp.*, vol. 18, no. 4, pp. 36-45, Jul. 2001

[12] Y.K. Malaiya, L. Naixin, J. Bieman, R. Karcich, B. Skibbe, " The Relationship Between Test Coverage and Reliability ", in *Proc. Softw. Rel. Eng. 5th Intl. Symp. on*, pp 186-195, Nov. 1994.

[13] J.Y. Jou and C.N.J. Liu, “Coverage Analysis Techniques for HDL Design Validation,” in *Sixth Asia Pacific Conf. on Chip Des. Lang. (APCHDL*99)*, Fukuoka, Japan, Oct. 1999.

[14] J. Voas, "Fault Injection for the Masses," *Computer*, vol. 30, pp. 129-130, Dec. 1997.

[15] G.A. Kanawati, N.A. Kanawati and J.A. Abraham, “FERRARI: A Flexible Software-Based Fault and Error Injection System,” *IEEE Trans. Comp.,* vol. 44, pp. 248-260, Feb. 1995.

[16] J.C. Baraza, J. Gracia, S. Blanc, D. Gil and P.J. Gil, “Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code,” *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 6, pp. 693-706, Jun. 2008.

[17] J.C. Baraza, J. Gracia, D. Gil and P.J. Gil, “Improvement of Fault Injection Techniques Based on VHDL Code Modification,” in *Proc. HLDVT*, 2005, pp. 19-26.

[18] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, J. Karlsson, “Fault injection into VHDL models: the MEFISTO tool”, in *Proc. of 24th Int. Symp. on Fault-Tolerant Computing (FTCS-24),* pp. 356-363, Austin, TX, USA, 1994.

[19] V. Sieh, O. Tschche, and F. Balbach, “VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions”, in *Proc. of Intl. Symp. On Fault-Tol. Comp. (FTCS'97).*

[20] N. Bombieri, F. Fummi, G. Pravadelly, M. Hampton, and F. Letombe, "Functional Qualification of TLM Verification", *in Proc. of ACM/IEEE Des., Aut. and Test in Europe (DATE) 2009* , pp. 190-195.

[21] S. Kim, J. Clark, and J. McDermid, “Assessing test set adequacy for object oriented programs using class mutation,” in *Proc. of Symposium on Softw. Tech. (SoST'99)*, pp. 72-83, Sept. 1999.

[22] S. Kim, J. Clark, and J. McDermid, “Class mutation: Mutation testing for object-oriented programs,” in *OOSS: Obj.-Orient. Softw. Sys.*, Oct. 2000.