MYGEN : Automata-Based On-line Test Generator for Assertion-Based Verification

Yann Oddos, Katell Morin-Allory, Dominique Borrione
TIMA Laboratory, Université Joseph Fourier
Grenoble, France
name.surname@imag.fr

Marc Boulé, Zeljko Zilic
McGill University
Montréal, Québec, Canada
marc.boule@mail.mcgill.ca, zeljko.zilic@mcgill.ca

ABSTRACT

To assist in dynamic assertion-based verification, we present a method to automatically build a test vector generator from a temporal property. Based on the duality between monitors and generators, we have extended the monitor generator tool MBAC to produce synthesizable on-line generators. We have tested the resulting generators in simulation and by emulation on an FPGA. The combination of multiple generators provides an efficient way to model the environment of modules within a DUT, facilitating an equivalent of software "unit testing" under real conditions, early in the design flow.

Categories and Subject Descriptors

B.2.3 [**Reliability, Testing, and Fault-Tolerance**]: Error-checking—*Test generation*; B.5.2 [**Design Aids**]: Verification

General Terms

Verification

Keywords

Test vector generation, semi-formal verification, PSL, generator

1. INTRODUCTION

Guaranteeing that a state-of-the-art system on chip (SOC) is exempt from design errors is a daunting challenge. Most emphasis is put on advancing the technology and improving the design methods. The relentless increase in the number of logic elements that can be placed on a single chip, combined with platform-based design, IP reuse, and design at higher abstraction levels, drive the complexity and performance increase exhibited each year. At the same time, despite significant progress in formal and dynamic verification techniques, the gap between the design and verification capabilities widens.

Dynamic verification, featuring simulation and/or emulation, remains the only viable solution for checking complete systems, but its quality critically depends on the thoroughness of both the system *specification* and the *test sequences*. A good specification should be mechanically interpretable in test scenarios, in order to ascertain that it correctly portrays the designer's intent. It should also be used in a wide variety of verification tools, all along the design and verification flow. Lots of progress in empowering designers to provide thorough specification has been recently achieved through the Assertion-Based Verification (ABV) paradigm [6]. In ABV, the essential logic and temporal properties of the design and its environment are expressed in a declarative form, using a formal property description language. At any time the properties can be added to the design description, or separately linked for emulation, simulation or model checking purposes.

Two IEEE standards are mainly used to write temporal properties: PSL (Property Specification Language) [7] and SVA (SystemVerilog Assertions) [14]. In the following, all the properties are written in PSL, but our method applies to SVA as well.

As an example, consider the following PSL property **P1**:

Property **P1** : always ($Start \rightarrow Req$ until Ack)

Property **P1** states: for each cycle where $Start$ is 1, a request Req should be produced and maintained active (Req=1) as long as the acknowledge signal Ack is not active (Ack=0).

The trace on Figure 1 satisfies property **P1**. Signal $Start$ is active at cycle ♯1 and Req is fixed to 1 during the same cycle. Then Req remains active up to ♯7. At ♯8, signal Ack takes value 1. After this cycle, all the constraints have been satisfied and the trace complies with **P1**.

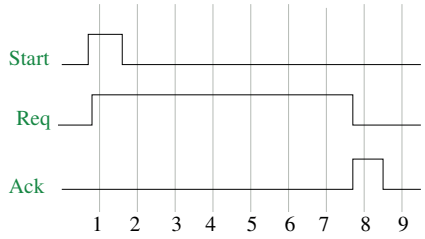

Figure 1: A trace satisfying property P1

Temporal properties are divided into two sets: assertions and assumptions. **Assertions** are used to specify functional aspects of the design under test (DUT). Assertions can be turned into *monitors* that check if the design is compliant with corresponding properties. **Assumptions** are used to constrain the behavior of the environment, which is very useful in restricting the *generators* of possible DUT input vectors. By avoiding non-compliant test vectors,

GLSVLSI'09, May 10–12, 2009, Boston, Massachusetts, USA.

the overall dynamic verification can be significantly faster, without impacting its quality.

While the verification with the help of assertions is widely supported in current model checkers (RuleBase, FormalCheck, etc.) and simulators (ModelSim, NCSim, etc.), modeling the environment of the DUT with formal assumptions is less widespread.

In this paper, we focus on the following problem in dynamic verification: how to automatically build an executable model of the environment, which generates test vectors that satisfy all assumptions about the environment. We present a simple method by which each assumption is first transformed into an efficient automaton, which is then turned into a synthesizable test generator. Please note that the main outcome here is an *on-line* test generator, which facilitates a high-throughput simulation/emulation when embedded with DUT, with no need for a separate test generation procedure.

The overall scenario that this outcome facilitates is as follows: starting with a HDL description of a design and the properties expressed in PSL, we provide a self-certifiable system, consisting of a test generator and the property monitors sufficient to check whether the properties are maintained across all self-generated test cases. The test generation and error reporting are well compacted: all that is needed is to check whether assertion monitors fire, signaling a failure. This scenario can be applied for testing modules (i.e. unit testing), or for complete designs, to improve verification productivity and quality.

Section 2 gives a brief overview of different approaches used to improve the quality of test vectors, by modeling the environment of the DUT at different levels. Our approach is described in section 3. The construction time and synthesis results on FPGA are given in section 4. Finally section 5 concludes on the method and its future extensions.

2. STATE OF THE ART

In the context of logic simulation and logic testing, a wide range of methods have been proposed to replace pure random test sequence generation by smaller, more focused sequences. Recently, assertion-based verification has renewed the interest in constrained test generation.

A first approach found in [15] applies the divide and conquer principle: the design is sliced into smaller components. Each one is used to extract constraints which are recombined to get the constraints for the whole design. Test vectors are obtained by analyzing these global constraints. Combined with an Automatic Test Pattern Generator (ATPG), all the random sequences not compliant with the interface specification are deleted, thus increasing the test efficiency.

An early method to produce test vectors compliant with a set of temporal properties combines the design with external blocks (one in Verilog and one in C) which dynamically constrain the inputs of the design [13]. The main drawback is the lack of expressive power due to the use of only one operator.

An automata theoretic approach is presented in [4]. Two automata are extracted from the system and the property, and the product automaton is constructed. State space traversal techniques are used to extract test vectors that lead to a failure. The method can suffer from a state explosion in the resulting product automata.

A two-step method is described in [11]. First, random vectors are applied, and the registers and wires that seldom or never change value are identified. Second, "input cubes" are computed, specially aimed at reaching these identified elements. The experimental results show very good test coverage results on small cases, but this gate-level approach may become complex on large circuits.

A black-box approach is presented in [10]. The goal is to increase the test coverage during simulation. From assertions on the design under test, the method produces satisfying test vectors without vacuity. The underlying formalization is a game between the environment and the DUT. Three different methods are defined depending on the level of observability and controllability of the signals involved in the assertion.

In contrast to all the above previous work, our approach only relies on the property formulas, and not on the DUT, to generate test vectors: PSL properties are synthesized into hardware modules that generate compliant signals. The same applies to the concept of "cando objects" [5], except that their solution is technically very different and oriented towards model checking rather than on-line execution; in particular, they do not support arbitrary repetition ('+' and '*') operators.

In [9], the HORUS tool has been developed to produce synthesizable and low complexity test vector generators from temporal properties. The idea is to define a library of primitive components (one for each PSL primitive operator), and an interconnection scheme to build a generator for a complex property.

The method presented in this paper is the result of a cooperation between the groups that developed MBAC and HORUS. The motivation was to benefit from the efficiency of MBAC in automata processing [3], and the duality between language generation and recognition, to unify the construction of generators and monitors.

3. GENERATORS WITH MYGEN

3.1 Preliminaries

Building Checkers from Properties.

The MBAC tool takes as input a temporal property and produces a monitor for a given PSL property [3]. To achieve this goal, it builds a non-deterministic automaton that recognizes all the traces that violate the property. Then, the automaton is transformed into a synthesizable RTL checker.

Basic automata have been defined for a kernel of primitive PSL operators. The remaining operators are translated in terms of the kernel, using rewrite rules. These rewrite rules have been proven correct with respect to the PSL semantics, using the PVS theorem prover [8]. Complex properties are turned into a monitor in two steps: apply rewrite rules as necessary, and combine basic automata to obtain the final automaton for the checker. Automata and logic optimizations are performed to minimize the size of the final checker.

The right part of Figure 2 depicts the automaton produced by MBAC for the following property **P2**:

Property **P2** : always ($sig1 \rightarrow$ next $sig2$)

Transitions are labeled with conditions, i.e. Boolean expressions built over a combination of the signals involved in the property. Each combination of an active state and input condition is evaluated to determine which next states are to be activated. The automaton then transitions into a new set of active states, and the process repeats at each clock cycle. Each time a final state is active, the property has failed and a counterexample trace can be analyzed.

Two different modes are used in MBAC to treat a property. A Boolean or sequence appears in two different semantic contexts, depending on how it is used in a property [3]:

- Obligation mode: Expression for which the failure of a sequence or Boolean expression must be identified. The automaton must match a Boolean expression or sequence when it fails to occur.

- Conditional mode: Expression for which the detection of a sequence or Boolean expression is required.

For example, in a property implication with two sequences (s_1 |–> s_2), a successful detection of s_1 enforces the obligation that s_2 must be matched, hence the two modes described above. The obligation mode version of a sub-automaton is obtained from the typical conditional mode automaton by applying an algorithm called `First_Fail`.

For the P2 example, Figure 2 illustrates how the second transition leads to a failure of the property (right side of figure). If at a cycle *n*, *sig1* is active, then the property fails if at cycle *n+1*, *sig2* is false. In the general case for more complex properties, `First_Fail` may modify the structure of the automaton.

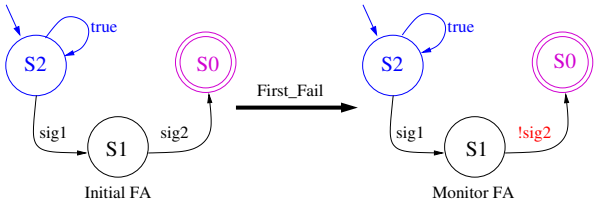

Figure 2: Monitor and Generator Finite Automata (FA)

From Monitors to Generators.

Generators and monitors are dual concepts: a monitor *recognizes* the complement language[1] of a property, while a generator *produces* the language of the property. This symmetry is depicted at the automaton level in Table 1. Since the monitors built by MBAC are of low complexity and efficient in emulation, we reuse the MBAC approach and automaton construction mechanism to produce synthesizable generators.

Table 1: Symmetry between Monitors and Generators

	Monitors	Generators
transitions' labels	observed	generated
final states	failure	satisfaction without vacuity

3.2 Construction of a Generator

Global description.

Figure 3 illustrates the steps required to build a generator from a property. First, the MBAC tool is run in generator mode and elaborates the generator automata as opposed to checker automata. Generators machines produce accepted traces for the property, as opposed to monitor automata which mark assertion failures. Finally, the synthesizable HDL description for the generator is extracted.

MBAC-generator.

The basic idea is to avoid the application of the `First_Fail` algorithm. The resulting automaton describes all the traces compliant with the corresponding property. This effect can be observed in Figure 2. The underlying concept is that the `First_fail` applies a form of negation on a portion of the automaton. This algorithm gives the set of all the traces not compliant with a property. By avoiding its application, we obtain the set of all the traces compliant with the property. The implementation in MBAC is actually more involved, and requires particular modifications to algorithms for operators such as "until" and "before".

[1]In dynamic verification, this is actually not the exact complement.

3.3 The Generator

Description of the Generator Automaton.

All generators have a generic interface. They take as inputs the synchronization signals *clk* and *reset*. Their outputs are composed of all the signals involved in the corresponding property.

The HDL description is composed of two blocks: one modeling the generator automaton and one used for pseudo-random number generation. As more than one state can be active during the same cycle, one flip-flop is used for each state. The HDL process for the FA block always has the same structure:

1. Selecting active states (line 1 of the HDL code in Figure 5);
2. Selecting outgoing transition(s) for the current active state (lines 2, 4 and 8);
3. Selecting values of output signals for the current transition (lines 9, 14, 18).

In every terminal branch of the automaton code (lines 3, 10, 15 and 19 in Figure 5), two types of actions are performed: output signal assignments, and activation (resp. deactivation) of destination (resp. source) states.

Random Numbers.

Several traces can satisfy the same property. Therefore, more than one path may lead to a final state verifying the property: there exist some states where a choice among several possible outgoing transitions has to be made. A pseudo-random block is used to randomly choose one transition.

The pseudo-random component is not only used to select a transition. Given a transition, its associated condition can be a complex Boolean expression, involving several signals: more than one valuation for the signals can satisfy the condition. The choice between these valuations is carried out by the pseudo-random block.

Two different kinds of random number generators were tested : Linear Feedback Shift Registers (LFSR), and specific Cellular Automata (CA) [16] suitable for random number generation such as CA30. CA are more efficient in terms of synthesis and random quality [12], but their characterization is complex and actually still incomplete.

The random block can be parameterized to modify the quality of the traces produced by the generator. Moreover, it is possible to build a generator using different modes provided by our tool. This enables more precise control over the kind of test vectors that are to be generated:

- **SimpleRand**: simple random generation shares the random component for different tasks (selects transitions for all the states, defines the random value of the signals etc...). The quality of the resulting traces is low due to the dependencies induced by this sharing of the random block; however, it can be useful for "quick-prototyping" on a small platform.

- **DirRand**: directed random generation allows the selection of the length of test vectors such as: shortest, short, medium, long etc... It can be used for an in-depth test of the design.

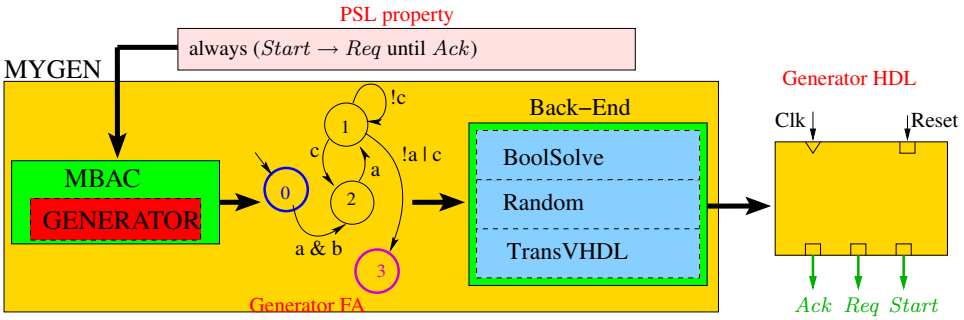

Figure 3: The Generator Flow - Embedding of MBAC into MYGEN

- **RealRand**: realistic random generation uses a distinct random block for each choice to be made in the FSM. The quality of test vectors is maximal and the generator covers all the set of traces complying with the corresponding property. This is particularly suitable for a complete test without excluding any possible corner case, at the cost of a significant overhead in generator size.

The flexibility provided by these different modes is useful, because it allows the creation of different instances of the same generator, each one dedicated to a different step in the test phase.

Let "random register" denote a register containing pseudo-randomly generated numbers. For the selection of the transitions, the random registers are denoted `Trans`. This is illustrated Figure 5 in lines 2, 4 and 8.

BoolSolve.

For each transition, the generator must produce a combination of signals validating the condition's transition. It is possible that several signals satisfy the condition. To enumerate all these possibilities and put them into the HDL description, a Boolean solver is used.

We use a simple and freely available tool called BoolSolve [1]. It takes as input a Boolean expression (the condition's transition in our case), and provides all the correct valuations of signals for this expression. All these possibilities are written into the HDL description. The random registers `Rand` are used to randomly select one of these valuations as it is shown in the HDL code in Figure 5, in lines 9 and 14. All the other signals (those not involved in the current condition's transition) have a random value.

Multiple Transitions.

The last issue to cover in order to build a correct generator is to take care of the dependencies that may exist between outgoing transitions from a given state. Consider the automaton in Figure 4, and assume transition `Trans(2)` is selected. Two scenarios may occur, depending on the values of signals (A,B):

- (0,1): in this case, nothing special happens. Trans(2) is active and within the next cycle, only S2 will be active.
- (1,0): here, Trans(1) is implicitly activated because A was assigned '1' for Trans(2). Then the two transitions are taken and within the next cycle, S1 and S2 will be active. That is why S1 is not deactivated for conditional branches (lines 14 and 18 in Figure 5).

These dependencies are analyzed by the MYGEN back-end prior to HDL code production.

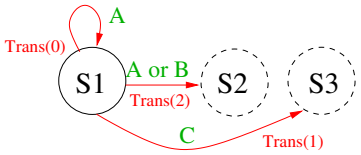

Figure 4: Part of a Generator FA

```
1. if S1=1 --Current State=S1
2.     if Trans(0)=1 --Cond=A
3.         A<=1;
4.     elsif Trans(1)=1 --Cond=C
5.         C<=1;
6.         S1<=0;
7.         S3<=1;
8.     elsif Trans(2)=1 --Cond=A or B
9.         if Rand(0)=1 and Rand(1)=1
10.            A<=0;
11.            B<=1;
12.            S1<=0;
13.            S2<=1;
14.        elsif Rand(0)=0 and Rand(1)=1
15.            A<=1;
16.            B<=0;
17.            S2<=1;
18.        else
19.            A<=1;
20.            B<=1;
21.            S2<=1;
22.        end if;
23.    end if;
24.end if;
```

Figure 5: HDL Code for the FA depicted in Figure 4

4. EXPERIMENTAL RESULTS

Experiments were conducted on a laptop equipped with a 2GHz dual core processor and 2 GB RAM, running a Linux Mandriva 2008 OS. The MYGEN is coded in C, and its efficiency is tested over a set of 60 properties which are partitioned into 3 groups:

- **PRIM**: this set contains the properties numbered from 0 to 27. They model primitive PSL operators (Boolean, Sequence and Foundation Language operators);
- **CPX**: properties numbered from 28 to 48. They model a variety of conventional PSL properties used in industrial cases;
- **LIM**: properties numbered from 49 to 59 are used to test the limits of the MYGEN tool. We have defined four types of properties: next[i], next_e[1..i], next_event[i] and [*i], with $i \in \{64, 128, 256, 512\}$.

All the properties are accessible on this web page: [2]. They are not described here for reasons of space.

The generators produced for these properties were used in simulation and served as a first validation of our approach.

4.1 MYGEN Analysis

Property complexity in the LIM group is one order of magnitude higher than in the other groups. Experimental results for the LIM are not displayed in Figures 7 and 6 in order to obtain clearer graphs.

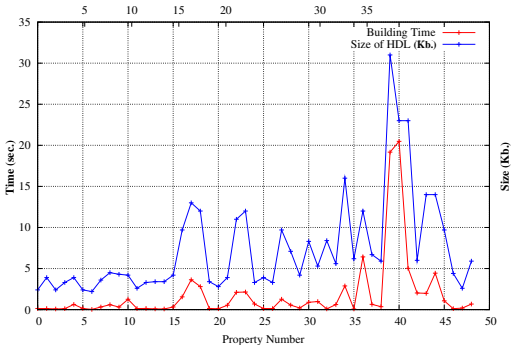

Figure 6: MYGEN Statistics : Run Times and HDL Code Sizes

Execution Time.

As can be seen in Figure 6, most of the generators are built in a few seconds. Almost all the time required to build a generator is consumed by the BoolSolve tool to list all the valid valuations for each transition condition.

Thus, the more transitions there are, the longer the generator construction time. For example, consider the following property Bench_{56} (belonging to the LIM group):

Property Bench_{56} :

always a $\rightarrow$ next_event_e(*sig1*)[1:512](*sig2*)

The generator's automaton has 510 transitions and it takes several minutes to extract the corresponding HDL description.

Generator Complexity.

While the construction time is related to the number of transitions in the automaton, the size of the HDL description depends essentially on the complexity of the transition conditions. The number of valid assignments for one transition condition can grow very sharply. As all the valid valuations for a transition are hardcoded in the HDL description, the size of the HDL description may also explode. Consider the following property Bench_{16} (belonging to the CPX group):

Property Bench_{16} :

always (sig1 or sig2 or sig3 or sig4 or sig5 or sig6 or sig7 or sig8);

The automaton for Bench_{16} is very simple and has only 2 transitions. One transition is labeled by the Boolean expression contained in Bench_{16}, for which $2^8 - 1$ valid assignments exist. The HDL description produced for this simple property has 4000 lines of source code.

These results also show that it is hard to anticipate the size of the generator just from the structure of the property: the size seems to be more influenced by the choice of Boolean and temporal operators.

4.2 Synthesis Results on FPGAs

Generators have been synthesized with QuartusII 7.2 on a CycloneII EP2C35 FPGA. Synthesis was performed with the synthesis options "balanced". Generators have been built in the mode RealRand of the MYGEN tool.

As shown in Figure 7, the complexity of the synthesized generators varies significantly. Figures 6 and 7 show that the construction time for the generator, the size of the HDL descriptions and the area used by generators are all related. The area, shown in the left graph of Figure 7, is nearly in linear relation to size of the HDL description, which is the blue curve in Figure 6.

Further, the size of the synthesized hardware generators can also increase dramatically, especially for properties from the LIM group. For example, the hardware generator for property Bench_{56} uses 122K logical cells and 70K flip-flops.

Finally, the left graph in Figure 7 shows that the amount of LCs and FFs for each generator is almost always equal. The resulting hardware thus has a very short critical path, enabling high clock rates, which attests to the efficiency of the proposed generators. Figure 7 shows that very good clock rates are indeed obtained. The minimum frequency is around 200 MHz, even for very complex generators. One can notice that the majority of the generators (60%) have the maximal frequency of 420 MHz imposed by the CycloneII technology.

5. CONCLUSIONS

A method has been defined to automatically build test vector generators from PSL properties. This approach, supported by the tool MYGEN, is completely independent of the DUT complexity, and can be used in a number of dynamic verification scenarios. The proposed methodology also frees verification engineers of the need to develop standalone test generation. Using a set of 60 properties, we have shown that the obtained generators are fast and relatively compact, which can be helpful in real design verification flows. In particular, our tool can produce a library of reusable parameterized components (SPI, OCP etc.), that can be instantiated with specific parameter values to fit the requirements for a specific test environment.

The automata theoretic approach that serves as a ground layer to process the properties is more efficient for monitors than for generators: the central mechanism in MBAC produces very compact monitors, but much bigger generators. A partial explanation lies in the FA simplification obtained from the `First_Fail` algorithm, used for monitors only. Also monitors are simple acceptors with a

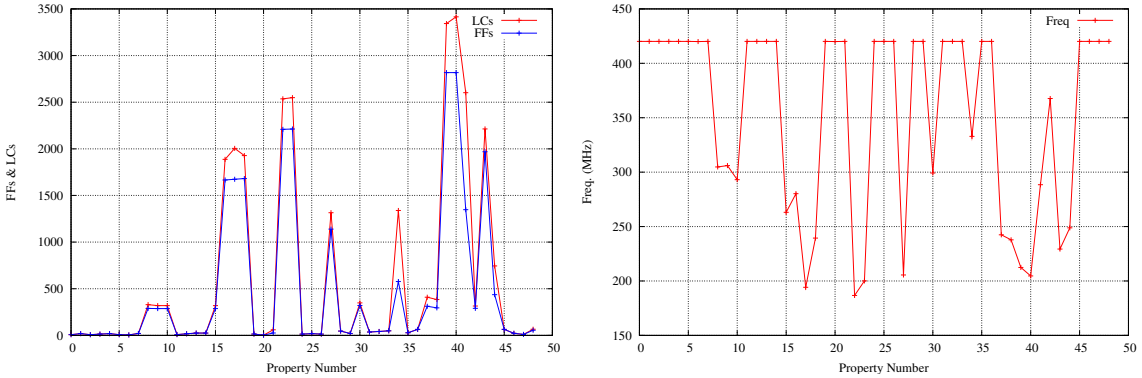

Figure 7: Synthesis Results: Area (left) and Clock Rates (right)

"yes/no" answer, while generators produce a set of possibly complex outputs, and require on-the-fly reasoning.

We also embedded into generators the pseudo-random components, which are portable (to ASICs and FPGA platforms, simulation engines, etc.), and significantly reduce the circuit size. If an external pseudo-random source is available, it can be directly used, removing the LFSR's or CA. The resulting generator size is then proportional to the number of states. In any case, the generator frequency is always very high. With our generators, we provide designers with a tool to build a model of the environment and test the design at-speed.

We are currently studying the random components (LFSR and CA) to improve the quality of the pseudo-random generation. From our experiments we conclude that CA are more efficient than the LFSR we tested. We would like to characterize a set of CA for our generators to automatically embed them into the generators. Further, the construction time for the generators can be improved by replacing the slow Boolean solver by a production quality SAT solver, e.g. MiniSAT. Finally, we plan a more complete verification by model-checking the generators and their corresponding properties. This will improve the confidence in the correct production of the synthesizable HDL code for the generators.

6. REFERENCES

[1] http://freshmeat.net/projects/bool-expr-solve/.

[2] http://www-tima.imag.fr/vds/horus/mygen_props/.

[3] M. Boulé and Z. Zilic. Automata-Based Assertion-Checker Synthesis of PSL Properties. *ACM Transactions on Design Automation of Electronic Systems (ACM-TODAES)*, 13(1):Article 4, January 2008.

[4] J. Calamé. Specification-based test generation with TGV. Technical Report R0508, Centrum voor Wsikunde en Informatica, May 2005.

[5] H. Eveking, M. Braun, M. Schickel, M. Schweikert, and V. Nimbler. Multi-level assertion-based design. In *5th ACM & IEEE International Conference on Formal Methods and Models for Co-Design MEMOCODE'07*, pages 85–87, Jun. 2007.

[6] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, Jun. 2003.

[7] H. Foster, Y. Wolfshal, E. Marschner, and I. . W. Group. *IEEE standard for Property Specification Language PSL*. pub-IEEE-STD, pub-IEEE-STD:adr, Oct 2005.

[8] K. Morin-Allory, M. Boulé, D. Borrione, and Z. Zilic. Proving and disproving assertion rewrite rules by automated theorem proving. In *IEEE International High Level Design Validation and Test Workshop 2008 HLDVT'08*, Nov. 2008.

[9] Y. Oddos, K. Morin-Allory, and D. Borrione. Assertion-based design with horus. In *6th ACM-IEEE International Conference on Formal Methods and Models for Codesign MEMOCODE'2008*, Jun. 2008.

[10] B. Pal, A. Banerjee, A. Sinha, and P. Dasgupta. Accelerating assertion coverage with adaptative testbenches. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 27, pages 967–972, May 2008.

[11] I. Pomeranz and S.-M. Reddy. Circuit lines for guiding the generation of random test sequences for synchronous sequential circuits. In *2008 conference on Asia and South Pacific Design Automation*, pages 641–646, 2008.

[12] B. Shackleford, M. Tanaka, R. J-Carter, and G. Snider. High-performance cellular automata random number generators for embedded probabilistic computing systems. In *In Proceedings of the 2002 NASA/NOD Conference on Evolvable Hardware EH'02*, 2002.

[13] K. Shimizu and D.-L. Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In *DAC*, pages 801–806, 2002.

[14] J. Srouji, S. Mehta, D. Brophy, K. Pieper, S. Sutherland, and I. . W. Group. *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language*. pub-IEEE-STD, pub-IEEE-STD:adr, Nov 2005.

[15] R.-S. Tupuri, A. Krishnamachary, and J.-A. Abraham. Test generation for gigahertz processors using an automatic functional constraint extractor. In *DAC*, pages 647–652, 1999.

[16] S. Wolfram. Random sequence generation by cellular automata. In *Advances in Mathematics*, volume 7, pages 123–169, 1986.