Towards an Efficient Assertion Based Verification of SystemC Designs

Ali Habibi and Sofiène Tahar
Concordia University
1455 de Maisonneuve, West,
Montreal, Quebec H3G 1M8
Email:{habibi,tahar}@ece.concordia.ca

Abstract— **In this paper, we present an approach to verify efficiently assertions added on top of the SystemC library and based on the Property Specification Language (PSL). In order to improve the assertion coverage, we also propose an approach based on both static code analysis and genetic algorithms. Static code analysis will help generate a dependency relation between inputs and assertion parameters as well as define the ranges of inputs affecting the assertion. The genetic algorithm will optimize the test generation to get more efficient coverage of the assertion. Experimental results illustrate the efficiency of our approach compared to random simulation.**

I. Introduction

SystemC [7] is among a group of system level design languages proposed to raise the abstraction level for System-on-a-Chip (SoC) design and verification. It is expected to make a stronger effect in the areas of system architecture, co-design and integration of hardware and software [7]. The verification of SystemC designs is a serious bottleneck in the design cycle. Going further in complexity and considering hardware/software systems will be out of the range of the used simulation based techniques. In fact, classical verification techniques when used with SystemC will face several problems related to the object-oriented (OO) aspect of this library and to the complexity of its simulation environment.

The main trends in defining new SoC verification methodologies are considering a hybrid combination of formal, semi-formal and simulation techniques. Assertions are set to be the next big breakthrough that will enable engineers to continue to design and verify larger and more complex designs. The Accellera Property Specification Language (PSL) [1] was developed in this respect to address the lack of information about properties and design characteristics in RTL modeling. It provides means of specifying design properties using a concise syntax with clearly defined formal semantics.

In this paper, we propose to augment the SystemC language to support the syntax and semantics of PSL. These latter are translated into external SystemC modules connected as read-only monitors (objects) to the original design. Every monitor is composed of a set of input signals (involved in the assertions) and a verification process (representing the code to verify the assertion itself).

The objective of the verification process is not only to write assertions but to verify them. This latter task is usually performed using test vectors generation tools mostly based on random processes. This kind of blind simulation does not guarantee that the assertion will be covered during the test execution. Therefore, it is very important to consider a smarter and more efficient test vector generation approach. To do so, we propose first to use static code analysis to extract a dependency relation between the design inputs and the variables considered in the assertion. This analysis will also define for every input the range of possible values that may affect the assertion which provides very useful information to improve the assertion's coverage.

In order to enhance the coverage even more, we also propose to use a genetic algorithm based on a community of random generators having a variety of DNA information [4]. This latter will help defining the list of variables considered in the test generation, their possible values and a weighted probability over the previous range. The DNA update/mutation rules will be defined according to the coverage each generator offers. At the end of the genetic procedure, we expect the final DNA to provide an identification of a generator that offers a better coverage than a random one.

The rest of this paper is organized as follows: Section II describes our approach to extend SystemC by PSL. Section III presents our methodology to improve the assertion coverage. Section IV illustrates our methodology on a Master/Slave Bus case study taken from the SystemC library. Section V discusses the related work. Finally, Section VI concludes the paper.

II. Extending SystemC by PSL

SystemC is a set of C++ class definitions and a methodology for using these classes [7]. SystemC introduces channels, interfaces, and events to enable communication and synchronization between modules or processes. An interface specifies a set of access methods to be implemented within a channel, where channels provide the implementation for these interfaces. An event is a flexible synchronization primitive that is used to construct other forms of synchronization. Events in SystemC occur at a given simulation time.

PSL is an implementation independent language to define properties. It does not replace, but complements existing verification methodologies like VHDL and Verilog test benches. The syntax of PSL is very declarative and structural which leads to sustainable verification environments. PSL consists of four layers based on the functionality [1]: modeling, verification, temporal and Boolean layers.

PSL is a hierarchical language, where every layer is built on top of the layer below. This approach allows the expressing of complex properties from simple primitives. A property (also

0-7803-8714-7/04$20.00 ©2004 IEEE

called assertion) is composed from three types of building blocks: Boolean expressions, sequences, which are themselves built from Boolean expressions, and finally subordinate properties. Sequences, referred to as SEREs (Sequential Extended Regular Expressions), are used to describe a single- or multi-cycle behavior built from Boolean expressions.

To add PSL assertions to SystemC two options are possible, namely, integrate the PSL as part of the library, or on top of the library. The former approach presents a radical change of SystemC requiring the addition of new constructors to the library (*assert* for example). Besides, the SystemC simulator and semantics must be updated in order to manage and verify the assertions correctly. Considering the OO aspect of SystemC and its modular structure, it is easier, yet probably more efficient, to add assertions on top of SystemC. In fact, any assertion can be seen as a monitor keeping track of some of the design signals, performing a verification operation and giving as output a status flag. The open question with this latter approach is how to update the design in order to connect the assertion's monitors.

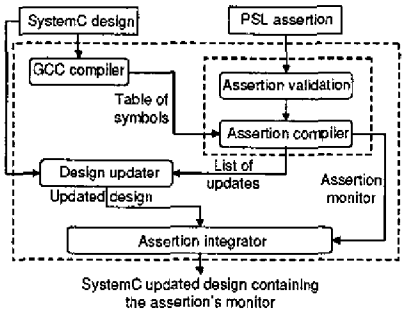

Fig. 1. Methodology of extending SystemC with PSL.

Figure 1 shows the proposed methodology to construct and integrate PSL into SystemC designs. We first start by collecting the information about the environment from the SystemC compiled code. We therefore, consider the symbol file generated from the Gnu-C-Compiler (GCC). This step is needed in order to localize which signals belong to which modules. Then, the assertion is validated and compiled. The validation phase verifies the syntax of the assertion while the compilation phase performs the link between the design variables and the assertion parameters.

In order to connect the assertion monitor to the design, this latter also needs to be updated. In fact, the signals involved in the assertion must be transformed to output signals in order to feed them to the assertion monitor. The list of signals required to extract from the design is generated by the assertion compiler then given to the design updater, which performs the required modifications to the original SystemC design. These modifications will not affect the behavior of the design since they will only get some signals connected to the assertion monitor as *read-only*. This latter is then connected to the updated design. When executed, the resulting code will therefore consider the assertion monitor as part of the design.

III. Assertions' Coverage Enhancement

Our goal is to define a test generation approach that offers better coverage of the assertions. To do so, we first start by statically analyzing the design in order to define a dependency relation between the system inputs and the assertions variables. Such a relation is very useful to omit the inputs that are not affecting the assertion. It serves also identifying the required inputs and the range of their possible values that may affect the assertion. We also identify which processes need to be activated in order to get the assertion fired. Figure 2 gives an overview of our methodology, including the following steps:

1. *Static Analysis*: We apply a static analysis technique to generate an abstract representation of the design modeled as graph, called *hypergraph* [10], that will include a representation of both the program's environment and the process's environment.
2. *Dependency check*: From the hypergraph representation, we extract the dependency graph and the range of inputs that may affect the assertion.
3. *Test Program generator*: Using the abstract program (modeled as a hypergraph structure) and the dependency graph, we generate a reduced model containing only the units involved in the assertion.
4. *Initial DNA generation*: Considering the list of input variables of interest for the assertion and their ranges, we create a DNA structure that will serve as starting point for the genetic algorithm.
5. *DNA evaluation/update*: Using the initial DNA, the algorithm will update the generators' community starting from the initial DNA to obtain an optimal DNA using the assertion coverage as selection criteria.

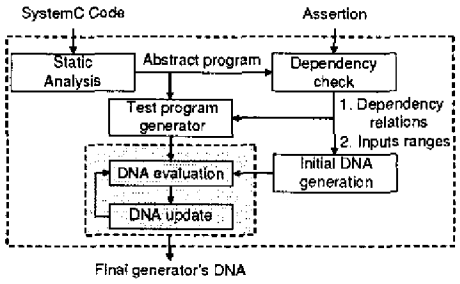

Fig. 2. Enhancing the Assertion's Coverage.

A. Static Code Analysis

In order to analyze SystemC designs statically and extract the required information to generate the "inputs/assertions variables" dependency relation, we considered an approach based on abstract interpretation [3]. Abstract interpretation is a formal technique that has proven to be efficient with object-oriented languages and large programs.

At the end of the analysis, the program is represented as a hypergraph [10], which can be interpreted as a general

automata connecting its states by branches (also called hyper-branches). These branches can be seen as an extension to Binary Decision Diagrams (BDDs), but more adapted to programs representation. We augmented this work to support the SystemC library and simulator in the form of specific classes to extract information related to SystemC processes and events from the design [5].

B. Genetic Algorithm

Genetic algorithms belong to a family of computational models inspired by evolution [6]. They encode a potential solution to a specific problem on a simple chromosomes like data structure and apply recombination operators to these structures to preserve critical information. Since their introduction by Holland [6], genetic algorithms have been applied to a broad range of learning and optimization problems [8]. Typically, a genetic algorithm starts with a random population of encoded candidate solutions (test generators for our case), called chromosomes. The objective is to maximize the likelihood of generating an optimal solution. This can be guaranteed by: (1) evaluating the *fitness* of each candidate solution in the current population; (2) selecting the fittest candidate solutions to act as parents of the next generation of candidate solutions; and (3) selected parents are recombined and mutated to generate offsprings.

In our context, the search space to be explored is the state space of the system that may trigger the assertion(s) under verification. Candidate solutions are finite sequences of input ranges and probability weights. Each candidate solution is encoded by a chromosome (a finite string of bits). The information encoded in the DNA includes: (1) the list of input variables, (2) their ranges (possible values), and (3) a weighted probability to their random generation. The algorithm evaluates the fitness of the candidate by executing a test generation based on the information embedded in the corresponding chromosome. A coverage report is then generated to serve in the fitness evaluation phase.

The chromosome encoding is the most important aspect of our algorithm. During the static analysis phase, we obtain the list of variables of the program and their types. Each variable is given a unique identifier. Each type is also given a space of possible values (for the type *char* for example the range is [0..255]). The chromosome encodes the list of variables, their types and a weight relation over the range of possible values. This latter varies according to the type and its interpretation. For every basic type, we defined a list of possible weight relations, e.g., for *Integer*, we use the following window relation:

$$I < -50 \text{ or } I > 50 \quad w = 0.2$$
$$-50 \leq I \leq 50 \quad w = 0.8$$

This relation states that the integer variable I is generated randomly in the interval [-50, 50] with a probability of 80% and 20% inside and outside the interval, respectively.

The proposed fitness function serves to guide the genetic search towards firing the assertion's variables. Its intuitive idea is to reduce the range of possible values of the input variables and to find the best probability distribution of the random test generation that will modify the assertion's variables. This way, we maximize the assertion evaluations, since the evaluation of the chromosomes is defined as an award bonus proportional to the number of assertion evaluations. In order to improve the efficiency of the algorithm, we keep track of the best and worst chromosome fitness in each generation; if both fitness values become equal, we increase the mutation rate, in order to help the genetic evolution get out of local maxima. Once there is an improvement in the overall fitness, we restore the original mutation rate to continue the evolution normally.

TABLE I
ASSERTIONS' COVERAGE ANALYSIS

Assertion	Rand. Test (%)	Init. of GA (%)	GA after 35 iter. (%)
A1	10	34	92
A2	8	42	93
A3	12	32	85
A4	11	37	89
A5	14	41	87
A6	16	46	91
A7	10	41	94
A8	17	33	83
A9	16	31	82
A10	14	45	97

IV. Master/Slave Bus Case Study

To illustrate the proposed SystemC verification methodology, we consider in this section a Master/Slave bus structure model taken from the SystemC library [1].

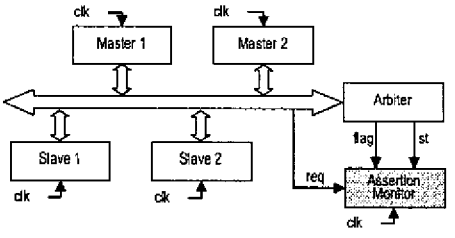

Fig. 3. Master/Slave bus structure.

Figure 3 shows the overall structure of the Master/Slave bus. Multiple masters can be connected to the bus via a communication interface. Each master is identified by a unique priority represented by an unsigned integer number. This structure includes several SystemC components and nicely takes advantage of the principles of using SystemC at the transactional level.

In order to evaluate the proposed genetic algorithm, we

[1] A more detailed description of this case study including all the source code is available at: http://hvg.ece.concordia.ca/Research/SoC/GeneticAlgo/. Other case studies are also available at the same web URL.

considered a set of 10 assertions[2]. Table I compares the assertion coverage results obtained: (a) in the initialization phase of the genetic algorithm (GA), i.e., just after the first DNA was generated from the static analysis phase; (b) after 35 generations of the GA; and (c) with a blind random generation. We used 10^9 simulation cycles for every generation. The coverage is measuring the portion of complete state space of the assertion covered by the test. We clearly notice that the static analysis phase already offers a better initial state than starting with totally random generation.

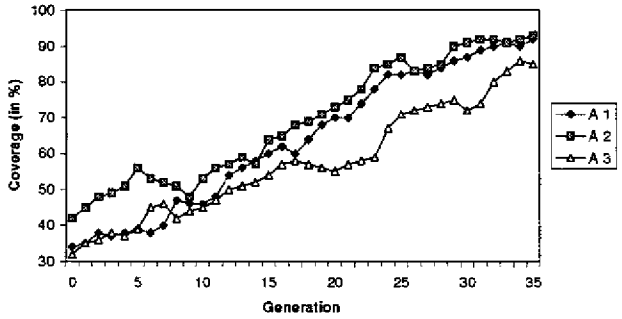

Fig. 4. Assertion coverage evolution as function of the Population Generation.

Figure 4 gives more details about the evolution of the algorithm for the three assertions (A1, A2 and A3). Typically, a genetic algorithm makes relatively quick progress in the beginning stages of evolution. We noted that there exist some phases, where the algorithm hits local maxima before mutating further, which improves its performance. We even noticed that the coverage sometimes decreases slowly from generation to generation (for e.g., generation 20 for A3). This is due to the fact that the evaluation of the assertion is based on weighted random generation. In other terms, since the number of tests is finite, a generator may have two different coverage results for two different test trials.

V. Related Work

Genetic algorithms have already been used for a broad range of applications. The most related work to ours is the one of Godefroid et *al.* [4], which in contrast to other approaches, addressed in particular the exploration of large state spaces of concurrent reactive systems as defined for model checking. Nevertheless, this work was restricted to simple Boolean assertions and was based on BDDs which is not suitable for high level languages like SystemC. We added to [4] a static analysis phase of the code before applying the genetic algorithm. We also considered a chromosome-encoding based on weighted probability over the space of the possible values of the program variables.

There exist a variety of very efficient EDA tools for test and assertion coverage, e.g., Specman Elite [11] from the Verisity, TestBuilder [2] from Cadence and TestBencher Pro [9] from SynaptiCAD. They use a user-defined constrained random simulation in order to perform higher functional coverage. Nevertheless, these tools do not take advantage from the design specific properties. Besides, they relate the coverage to the number of times the assertion was executed while a correct evaluation has to consider what portion of the assertion's state space was covered. For instance, actual tools were defined for low HDL level designs (using Verilog and VHDL) and do not define coverage metrics for PSL assertions when used with SystemC. We are not aware of any other work where genetic algorithms have been combined with static code analysis to optimize test vector generator in order to improve PSL assertions coverage with SystemC. As future work, we consider to implement our benchmarks in Verilog in order to be able to evaluate partially the performances of our approach in comparison to existent testbench tools.

[2]Due to the lack of space in the paper, we refer the reader to http://hvg.ece.concordia.ca/Research/SoC/GeneticAlgo/ in order to get a detailed description of the assertions.

VI. Conclusion

In this paper, we presented a methodology to integrate PSL with the SystemC language. We proposed to translate PSL into SystemC monitors connected to the design in order to verify some assertions during simulation. Our approach takes advantage from the OO nature of the C++ language and the events concept of the SystemC library. In order to verify efficiently assertions in SystemC, we further apply a static code analysis technique based on abstract interpretation. This phase generates an abstracted version of the initial design modeled as a hypergraph that helps defining the dependency between the system inputs and the assertion's variables, as well as restricting the possible values of the inputs to certain ranges that may update the assertion. Although experiments showed that this approach improves the assertion's coverage, we proposed to use a genetic algorithm that optimizes the probability distribution of the inputs over the space of their possible values. Our genetic algorithm showed an improvement of the assertions coverage by a factor of eight in comparison to the random case. As future work, we target to optimize the genetic algorithm to improve various coverage metrics.

References

[1] Accellera Organization. *Property Specification Language Reference Manual.* 2003.

[2] Cadence Design Systems. *Cadence Verification Extensions, V. 5.0.* 2003.

[3] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[4] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Tools and Algorithms for Construction and Analysis of Systems*, 2002.

[5] A. Habibi and S. Tahar. Abstract interpretation of systemc designs. Technical report, Department of Electrical and Computer Engineering, Concordia University, June 2004.

[6] J. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, 1975.

[7] Open SystemC Initiative. Website: http://www.systemc.org, 2004.

[8] H. Rudin. Protocol development success stories: Part i. In *12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Florida, USA, June 1992.

[9] SynaptiCAD Inc. Website: http://www.syncad.com/, 2004.

[10] F. Vederine. *Analyses totales de programmes par interpretation abstraite.* PhD thesis, Ecole Polytechnique, Paris, France, 2000.

[11] Verisity Ltd. Website: http://www.verisity.com/, 2004.