Combining System Level Modeling with Assertion Based Verification

Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal
IBM Haifa Research Lab, Haifa Israel
Lyes Benalycherif, Romain Kamdem, Younes Lahbib
ST Microelectronics, Grenoble, France

Abstract

Assertion-Based Verification (ABV) using the PSL language is currently gaining acceptance as an essential method for functional verification of hardware. A basic technique to implement ABV is to embed temporal assertions in RTL code. This paper describes the use of a PSL-based ABV methodology in a C++-based system level modeling and simulation environment. We describe the considerations of porting a tool which translates PSL to VHDL/Verilog, to support C++, a language which was designed for software and does not have concurrent language constructs. The translation scheme is shown to be adaptable to all C-based environments. We exemplify the wide applicability of this scheme by detailing its successful deployment in a SystemC-based industrial System-on-Chip (SoC) project.

1. Introduction and Motivation

As the complexity of hardware designs has grown to the degree that the traditional approaches have limitations, the need for a better verification methodology, one with improved levels of observability of the design behavior and controllability of the verification process has become clear. Assertion-Based Verification (ABV) has been identified as a modern, powerful verification paradigm that can assure enhanced productivity, higher design quality and, ultimately, faster time to market and higher value to engineers and end-users of electronics products. With ABV, assertions are used to capture the required temporal behavior of the design, in a formal and unambiguous way. The design then can be verified against those assertions using simulation and/or static verification (e.g. model checking) techniques to assure that it indeed conforms to the intended design intent, as captured by the assertions.

ABV has gained a very strong momentum over the last years, as evident by the increasing number of verification of experience reports and case studies, books, commercial offerings, and standard activities and academic research efforts in this area. A significant ingredient of this momentum has been the official selection of the language PSL [3], based on the Sugar language from IBM, as an industry standard. Over 50% of the respondents of a recent survey conducted by John Cooley, on the proliferation of ABV [4] said they use or plan to use ABV on their next project. Of those that responded positively, PSL was by far the most popular assertion language of choice.

In most industrial settings, an evolutionary approach to ABV has been taken where first and foremost assertions are used as a part of the traditional simulation methodology – with which most engineers feel more comfortable with. In accordance with this tendency, and to enable effective deployment of ABV in a simulation environment, we have developed a platform called FoCs ("Formal Checkers") [7]. FoCs takes PSL/Sugar properties as input and translates them into assertion checking modules ("checkers") which are integrated into the simulation environment and monitor simulation on a cycle-by-cycle basis for violations of the property. For each property of the specification, represented as a PSL Property, FoCs generates a checker for simulation. This checker essentially implements a state machine which will enter an error state in a simulation run if the formula fails to hold in this run. In Section 3 we will describe the checker generation process in more detail.

The philosophy behind the development of FoCs has been to provide enhanced productivity to simulation through automation of the checker creation effort. Checkers are, in fact, a traditional part of simulation environments ([9]). They facilitate effective testing, as they automate test results analysis. Moreover, checkers facilitate the analysis of intermediate results, and therefore save debugging effort by identifying problems directly - "as they happen", and by pointing more accurately to the source of the problems. However, the manual development and maintenance of checkers is a notoriously high-cost and labor-intensive effort, especially if the rules to be verified are complex temporal ones. This has been an impeding factor to verification productivity. For instance, in the case of a checker for a design with overlapping transactions (detailed in Section 3), writing a checker manually is an excruciating error-prone effort.

Several solutions have been proposed over the last five years to avoid the inefficiencies involved with manual checker writing. Generally speaking, these approaches include libraries of checkers such as the popular public-domain OVL library [2] and proprietary libraries such as CheckerWare from 0-in [1]. These libraries serve as repositories of checking modules which can be instantiated as necessary in test-benches and facilitate specific checking goals. For example, OVL has a module for checking that a specified expression must not change its value for a specified period of time; and CheckerWare has modules for checking various arbitration policies. While very useful when applied for checking specific situations, the largest deficiency of libraries such as OVL is that they are fixed-form and thus generally inflexible (though some of them offer a certain degree of configurability).

Observing the inefficient process of manual checker writing in ongoing projects, and the limited remedy offered by libraries of checkers, we recognized that for best results, and effective ABV solution needs to include two components: a formal declarative language for defining the desired checks, and a mechanism for automatically transforming the definition of the checks into effective checking modules. This has inspired the development of FoCs as a means for automatically generating checkers from simple specifications. For example, SystemVerilog [12] also has a set of language constructs for writing assertions, called simply SystemVerilog Assertions (SVA). A similar approach to ours is proposed by [10] to augment SystemC with SVA but only considering a subset of SVA and relying for the translation of SVA into SystemC modules on internal development.

SystemC [5] was developed about six years ago as the verification platform for SoC verification and hardware-software co-verification. SystemC is a set of C++ libraries that let designers and verification engineers write hardware like modules in C++. SystemC contains classes that imitate low level hardware constructs like registers and also higher level hardware constructs like channels. Despite the energy and momentum that vendors and chip developers put into SystemC, the acceptance of SystemC as a way to implement verification was slower than expected. However, the last two years have seen a growing interest in the platform since SoC development is on the rise and is becoming more popular in design starts [11].

FoCs has been put to multiple types of successful use by engineering teams in the industry. In those past project, FoCs was used to translate the assertions directly to native language of the design (VHDL or Verilog) which have built-in temporal and concurrent semantics. Generation Checkers for SystemC required the definition of PSL semantics for C++. Instead of developing a PSL to SystemC translator, we developed a translation mechanism that is suitable for any C++ based modeling environment and not necessarily SystemC. We used this version of FoCs to implement PSL based ABV on a industrial System-on-Chip (SoC) which had a SystemC high-level.

The rest of this paper is as follows. Section 2 gives a brief overview of PSL. Section 3 describes our translation method to C++. In Section 4 we describe the SoC and the results of using FoCs for implementing PSL based ABV on top its SystemC based verification platform. The final section details our conclusions and future plans.

2. The PSL Language

PSL/Sugar is a language for the formal specification of hardware. It is used to describe properties that are required to hold of the design under verification. For instance, it might be required that "if a **`request`** is made, it must stay asserted **`until`** a **`grant`** is received”. This simple English specification leaves a lot to be desired: Which signals indicate a request, and which indicate a grant? By **`until`** do we mean that the request signal must stay asserted only up to, but not including the cycle in which a grant is received, or must they also stay asserted the cycle of the grant? These questions could of course be answered easily in English. But for complicated specifications, using English to explain exactly what is meant can be difficult. PSL provides a means to write specifications which are both easy to read and mathematically precise. PSL became an Accellera standard in 2002 and is currently ongoing a standardization process in IEEE (IEEE 1850 working group).

PSL consists of four layers:

1. The Boolean layer is comprised of Boolean expressions. For instance, **`a`** is a Boolean expression, having the value `1` when signal **`a`** is high, and `0` when signal **`a`** is low. PSL interprets a high signal as `1`, and a low signal as `0`. The Boolean layer also contains Boolean operators such as and, or, xor, etc.
2. The temporal layer consists of temporal expressions or properties which describe the relationships between Boolean expressions over time. For instance, **`always (req ->next ack)`** is a temporal property expressing the fact that whenever (**`always`**) signal **`req`** is asserted, then (**`->`**) at the next time (**`next`**), signal **`ack`** is asserted. In Section 2.1 we explain how time progresses in PSL.
3. The verification layer consists of directives which describe how the temporal expressions should be handle by the verification tools. For instance, **`assert always(req -> next ack)`** is a verification directive that tells the tools to verify that the property holds. Other verification directives include an instruction to assume, rather than verify, that a particular temporal property holds, or to specify coverage criteria for a simulation tool. The verification layer also provides a means to group PSL statements into verification units.
4. Finally, the modeling layer provides a means to model behavior of design inputs, and to declare and give behavior to auxiliary signals and variables. The modeling layer is also used to give names to properties and other entities from the temporal layer.

2.1 The Passing of Time in PSL

PSL contains a temporal logic to reason about discrete time systems (hence the **`next`** operator). However, although PSL has the ability to specify temporal relationships, it does not explicitly define how time progresses. The designers of the PSL language made it the responsibility of the tool developer to define how time progresses. For instance, a cycle-based simulator defines time progression strictly on clock cycle boundaries, while an event-based simulator defines time progression to occur on every change of a signal value. In fact, the same PSL assertion may evaluate differently on each of those tools (it may pass on one and fail on another). In order to overcome this, PSL gives the user the ability to override the tool definition and explicitly define the temporal expression by "clocking" the temporal property (see PSL LRM [3] for details). This is useful when a user wants to reuse PSL code in different tools.

When the IBM FoCs tool is used to translate PSL to VHDL or Verilog it assumes that the user will supply an external signal which will mark the progression of time every time its value rises from 0 to 1. For C++, we will see in the next section that FoCs completely delegates the decision on how time passes to external logic.

3. From PSL to C++

Roughly speaking, the FoCs tool takes an assertion and generates a state machine that monitors a set of signals and performs an action whenever a final state is reached. Figure 1 shows the overall environment in which FoCs operates. The user provides a design to be verified, as well as a formal specification written in PSL and a set of test programs generated either manually or automatically. FoCs translates the formal specification into checkers, which are then linked to the design and simulated with it. During simulation, the checker produces indications of property violations. It is up to the user to decide what action to take: fix the design, the property, or the simulation environment. In addition to checkers, FoCs can also be used to take PSL coverage statements in order to generate coverage monitors that report appropriate coverage information.

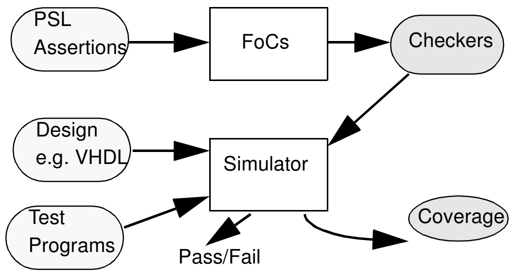

Fig. 1. The FoCs Flow

FoCs translates PSL into the target language – which can be VHDL, Verilog, or C++ – in the following way: first, each property is translated into a nondeterministic finite automaton (NFA). The NFA has a set of distinguished error states, and the property specifies that the NFA will never enter an error state (entering an error state means that the design does not adhere to the specification under the test conditions).

To be of any practical use, the NFA has to be converted into a deterministic automaton (DFA). The number of states of the DFA may be exponential in the number of states of the NFA, but simulation is sensitive to the size of the representation (the number of lines in the checking module) rather than to the number of states. Practically, it is almost always linear because of the types of properties that people tend to write.

Once the generation of the DFA is done then FoCs translates to the target language, Verilog, VHDL or in the case of this work C++. This translation is completely decoupled from the DFA generation. The translation details are described in [7].

Example: The following example will demonstrate the conciseness and ease of use of PSL for checker writing and the advantage of automatic checker generation. Assume that the following property is part of the specification: If a transaction that starts with tag **`t`** has to send **`k`** bytes, then at the end of the transaction (identified by the tag **`t`**) **`k`** bytes have been sent. The user can formulate this property in PSL as follows:

```
forall t in [1:4]
  forall k in [1:8]
    assert {[*]; trans_strt && tag = t
            && bytes_to_send= k; true[+];
            trans_end && tag=t}
                       (bytes_sent=k);
```

Manual writing of a checker for this property may become complicated if transactions may overlap, which means that a new transaction may start while previous transactions are still active. The checker writer has to take into consideration all possible combinations of intervals and perform non-trivial bookkeeping. The PSL formula is evidently much more concise and readable than the resulting VHDL entity, Verilog module or C program.

3.1 Translation to C++

In order to translate to C++ FoCs needs four additional inputs on top of those depicted in Figure 1.

a. Mapping file – every assertion includes signals which need to be mapped to the real design signals. The users create a mapping file, which indicates for each signal to which signal in the design it is connected and its type.

b. Report template – the checker's output template file, written in the target language (C++) using special FoCs directives (it is not mandatory and not detailed in this paper).

c. Adaptor template file – the vehicle in which the user customizes the C++ checker output to his simulation environment - described below.

d. FoCs settings file – the file which contains specific per-checker settings which provide additional information necessary for the checker generation.

Since C++ does not have a natural definition for time progression the design decision was to generate C++ code separated into two different parts:

1. The checker logic (state machine transition function and the assertion condition), generated from the PSL assertion. The checker is implemented as class with a **`reset`**, **`transition`** and **`mapping`** methods: the **`reset`** method initializes the checker internal signals, the **`transition`** method performs the needed computations of the next state, based on the current state, inputs and the transition function and the **`mapping`** method receives port name and value and assigns the value to the port, when **`port`** is a signal which behavior is undefined and the user must connect it to the signal in the design.

2. The checker control code. This is the Adaptor file which contains code to call the checker's reset and transition functions. The Adaptor file is generated from the user defined Adaptor template. The Adaptor template is C++ intermixed with special FoCs directives which provide handles to FoCs generated names used in the checker, which are not known at the time that the template is written. The user has to define when he wants to invoke the checker's reset and transition functions in the Adaptor template, and also when to invoke simulator specific functions which sample the design signal values in order to assign them to the corresponding checker ports.

In Figure 2 is an adaptor template file for systemC. It contains a generic **`wrapper`** class for the FoCs generated checker.

`@FOCS_CHECKER_CLASS@` in the wrapper class name on line **`2`** is replaced eventually by the FoCs checker class's type. The data members of the wrapper are defined on lines **`6-13`**. The directive **`@FOCS_CHECKER_CLOCK@`** on line **`6`** is replaced by the clock name that is defined in the FoCs settings. For each port, code that is written between the directive **`@FOCS_PORT_REPEAT_SECTION_BEGIN@`** and the directive **`@FOCS_PORT_REPEAT_SECTION_END@`**, is replicated. Between any two replications, FoCs puts a comma (,). On line **`9`**, the **`@FOCS_PORT_ALIAS@`** directive is replaced by the current replication port name. Thus, the net effect is that all port names are printed out with a comma separator in between each two. Figure 2 is a simplified version of the Adaptor template where all ports are defined as the systemC type **`sc_in<bool>`** but the full Adaptor template supports, all other port types by using the type information from the mapping file.

```
1 #include @FOCS_CHECKER_H_FILE@
2 class wrapper_@FOCS_CHECKER_CLASS@ :
3   public sc_module
4 {
5   public:
6      sc_in_clk @FOCS_CHECKER_CLOCK@;
7      sc_in<bool>
8  @FOCS_PORT_REPEAT_SECTION_BEGIN
9      @FOCS_PORT_ALIAS@
10 @FOCS_PORT_REPEAT_SECTION_END@;
11    int @FOCS_PORT_REPEAT_SECTION_BEGIN@
12      tmp_@FOCS_PORT_ALIAS@
13 @FOCS_PORT_REPEAT_SECTION_END@;
14    @FOCS_CHECKER_CLASS@ *instance;
15
16 void transition();
17 void update();
18 SC_CTOR(wrapper_@FOCS_CHECKER_CLASS@) {
19   SC_METHOD(transition);
20   sensitive_pos << @FOCS_CHECKER_CLOCK@;
21   SC_METHOD(update);
22   @FOCS_PORT_REPEAT_NL_SECTION_BEGIN@
23       sensitive << @FOCS_PORT_ALIAS@
24   @FOCS_PORT_REPEAT_NL_SECTION_END@
25   instance = new @FOCS_CHECKER_CLASS@;
26 }};
27
28  void wrapper_@FOCS_CHECKER_CLASS@::
29   transition(){
30    instance->transition(
31      @FOCS_PORT_REPEAT_SECTION_BEGIN@
32      tmp_@FOCS_PORT_ALIAS@
33      @FOCS_PORT_REPEAT_SECTION_END@);
34      if (!instance->getStatus())
35         cout <<
36            "@FOCS_CHECKER_CLASS@ failed";
37  }
38  void wrapper_@FOCS_CHECKER_CLASS@::
39   update(){
40    @FOCS_PORT_REPEAT_NL_SECTION_BEGIN@
41    tmp_@FOCS_PORT_ALIAS@ =
42         (int)@FOCS_PORT_ALIAS@.read()
43    @FOCS_PORT_REPEAT_NL_SECTION_END@
44 }
```

Fig. 2. An Adaptor Template for SystemC

Lines **`11-13`** define variables to convert the ports to integers in order to pass the port values to the checker t**ransition** function, which accepts integer parameters. Finally a pointer to the FoCs checker is defined on line `14`.

Lines `16-17` define the wrapper `transition` and `update` methods and lines `18-26` define the constructor.

In the constructor, the clock input is defined as edge sensitive on line `20,` and the rest of the ports are defined as level sensitive on lines `22-24`. This is done using the directives: `@FOCS_PORT_REPEAT_NL_SECTION_BEGIN@…@FOCS_PORT_REPEAT_NL_SECTION_END@`.Again, code between these two directives is replicated for each port but the separator is a semi-colon (;). The FoCs checker is dynamically allocated on line `25`.

Lines `28-37` contain the body of the wrapper `transition` routine. It calls the FoCs checker `transition` function (using the template loop directives to pass all the ports as parameters) and then checks for a fail on line `34`. If there is a fail, it prints an error message.

Lastly, lines `38-44` contain the body of the wrapper `update` routine. This routine uses the template loop directives to generate code that assigns updated values to all the `tmp_` integer variables.

Note that there are template directives not given in this example, to handle a reset signals and other requirements.

4. Using PSL to Verify a SoC

STMicroelectronics SoCs are articulated around a system bus with several processor cores, a DSP and a large number of complex IP blocks of dedicated logic such as decoders (e.g. MPEG), specific functions (e.g. modem), and standard/ proprietary IO functions (e.g. USB). These HW components come along with embedded SW such as an OS, drivers, communication and multimedia applications, etc. Both the complexity and the mixed HW/SW nature of the SoC, impose a system level design flow based on SystemC; the SoC development team undertakes a high-level modeling and verification effort early in the whole development process. SystemC models are built for each HW component and grouped into simulation platforms according to different abstraction levels: functional, transactional level (TLM) and Bus Cycle Accurate (BCA). The engineers that build these platforms as well as the ones who use them, are daily exposed to integration, IP reuse, debug and functional coverage. These observability and controllability issues are in fact, not specific to reasoning at system level and also exist at RTL. But because SystemC involves C++ types and constructs, these issues are more difficult to tackle and the challenge to overcome them is higher.

PSL plays a major role in verification and reuse of verification IP across the different verification platforms of the design flow. PSL assertions are part of the dynamic verification at RTL level and system level. The assertions are also mapped into an emulator and run at system speed using their synthesizable HDL views generated by FoCs. FOCS checkers are combined to RTL and submitted to SW-based simulations when using simulators not supporting PSL. In addition to that, PSL assertions have been exhaustively checked statically at block level using the RuleBase formal verification tool[8]. The ability to translate PSL to C++ enabled the deployment of these and other assertions in the SystemC simulation platform; they are translated into C++ checkers by FOCS and wrapped into SystemC modules that are later on connected as read-only monitors.

4.1 Case Study

Our SystemC BCA level model of the SoC was verified on the SystemC BCA platform. The SoC was composed of 5 blocks:

1. SH4 - a processor that executes C test programs and loads or stores) across the interconnect to the local memory controller (LMI).
2. TG - a traffic generator that sends dummy data to saturate the bus
3. A 32 bits STBus interconnect
4. FEMI - an external memory controller and its associated ROM that shelters the code run by the processor
5. LMI - the local memory controller and its DR-SDRAM

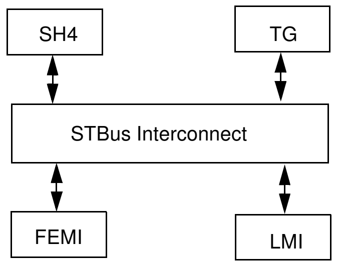

Fig. 3. Block Diagram of the SoC

The main purpose of this SystemC BCA platform was to perform architectural performance evaluation, analyzing the latency between the processor and the LMI, the bus occupancy of transactions to the LMI, the throughput of transactions to the LMI and the bandwidth allowed by the LMI. The other purpose of this platform was to validate the SystemC BCA models to be reused in other platforms for different SoCs. We focused for this validation on the interfaces of the SystemC BCA models and in particular on the adherence to the bus protocol since its rules and signaling information are implemented by the SystemC BCA abstraction level.

For checking the STBus, we deployed a library of PSL assertions specifying the protocol rules. These same assertions are largely used to check the adherence to the protocol at the RTL level. The SystemC platform was augmented with the PSL assertions involving the STBus signals of each of the SystemC BCA model.

For example, The STbus protocol specifies for the LMI interface that "if grant is asserted, it must remain asserted until request is asserted. This is expressed by the following PSL assertion:

```
  assert always((lmi_gnt & !lmi_req)
                 ->next lmi_gnt);
```

where **`lmi_gnt`** and **`lmi_req`** are **`sc_signals`** (systemC signals) of the LMI STBus interface.

The PSL assertions are embedded in the systemC code as smart comments and are picked up by a perl script from the systemC code and then copied in a separate file. The assertions file is then submitted to FOCs, which uses the extended version (supporting all port types) of the Adaptor file in Figure 2, to generate for each assertion a C++ checker and wrapper. The so generated FOCS code is attached to the SystemC platform as a read-only monitor. The wrapper corresponding to the assertion above is:

```
class wrapper_TM1 : public sc_module {
      public:
      sc_in_clk clk;
      sc_in< bool>lmi_req, lmi_gnt;
      int tmp_lmi_req, tmp_lmi_gnt;
      TM1 *instance;

      void transition();
      void update();
      SC_CTOR(wrapper_TM1) {
        SC_METHOD(transition);
        sensitive_pos << clk;
        SC_METHOD(update);
        sensitive << lmi_req;
        sensitive << lmi_gnt;
        instance = new TM1;
}};

void wrapper_TM1::transition()
{
  instance->transition(
        tmp_lmi_req,tmp_lmi_gnt);
  if (!instance->getStatus())
   cout << " TM1 failed ";
}
void wrapper_TM1::update()
{
  tmp_lmi_req = (int) lmi_req.read();
  tmp_lmi_gnt = (int) lmi_gnt.read();
}
```

To complete the picture wrapper instantiation and initialization code is inserted in the appropriate systemC **`sc_module`** (in this case the top level), as follows:

```
wrapper_TM1 *inst_wrapper_TM1;
    inst_wrapper_TM1->clk(Clock);
    inst_wrapper_TM1->lmi_req(LMI.req);
    inst_wrapper_TM1->lmi_gnt(LMI.gnt);
```

4.2 Results and Discussion

This platform which represents about 27000 lines of SystemC code has been simulated under Linux on a dual processor Pentium Xeon 3 GHz. Two assertions out of a total of 160, failed during simulation. The failures concerned the STBus interface of the SH4 processor and revealed a bug according to which 2 control fields of the protocol signaling were not correctly handled in the response pathway. The SystemC BCA model of the SH4 model has been corrected accordingly, permitting the architecture evaluation to be pursued. The test programs which ran on the SH4 processor during simulation involved a total of **3200000** transactions, which were simulated in 9 seconds with the assertions disabled. The assertions increased simulation time by a factor of 2.

5. Conclusion and Future Plans

By using FoCs, we found the translation of PSL to SystemC very simple and straightforward.

If we exclude the simulation time increase, PSL assertions on SystemC BCA were as beneficial as expected permitting to find a bug, localize it and fix it very quickly. In the near future, we plan to optimize the instantiation of the assertions by recognizing similar generic assertions and keeping only one C++ class for similar assertions. Further benefits from using PSL assertion in SystemC can be achieved by considering assertions beyond the SystemC BCA models interfaces to cover internal SystemC implementation aspects. It is also envisaged to augment with PSL, SystemC higher abstraction levels such as the TLM level.

References

[1] CheckerWare: http://www.0-in.com/products_monitors.html
[2] Open Verification library: http://www.eda.org/ovl/
[3] Property Specification Language: *Reference Manual*. Version 1.1, Accellera, June 2004.
[4] PSL Survey:http://www.deepchip.com/items/dvcon04-06.htm
[5] SystemC: http://www.systemc.org/
[6] [VRMIEEE Standard VHDL Language Reference Manual, ANSI/IEEE Std 1076-1993, published by IEEE.
[7] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, Y. Wolfsthal FoCs: *Automatic Generation of Simulation Checkers from Formal Specifications*. CAV 2000: 538-542
[8] I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman,, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, Y. Wolfsthal R*uleBase: Model Checking at IBM,* CAV 1997.
[9] D. Geist, G. Biran, T. Arons, Y. Nustov, M. Slavkin, M. Farkash, K. Holtz, A. Long, D. King, S. Barret, "*A Methodology for Verification of a System on Chip*", Proc. DAC'99.
[10] A. Habbibi, S. Tahar, On the Extension of SystemVerilog Assertion, CCECE/CCGEI 2004 May Niagara Falls, IEEE.
[11] G. Smith, "*The New SoC Economy or It's the Gates Stupid*", Dataquest Gartner, DAC 2004.
[12] S. Sutherland, S. Davidmann, P. Flake, P. Moorby, "*System Verilog for Design: A Guide to Using System Verilog for Hardware Design and Modeling*", Kluwer, 2004.