On the Extension of SystemC by SystemVerilog Assertions

Ali Habibi and So£ène Tahar
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve, West,
Montreal, Quebec H3G 1M8
Email: {habibi,tahar}@ece.concordia.ca

Abstract— **In this paper, we present an extension to the SystemC library by SystemVerilog assertions. SystemC is an emerging system level design and veri£cation language based on C++ object oriented paradigms. It enables the modelling and simulation of a complete System-on-a-Chip. We propose to extend the SystemC library with Assertion Based Veri£cation (ABV) which is a higher abstraction mechanism that allows a concise capturing of design speci£cation. In order to do so, we consider the same ABV structure as de£ned for the SystemVerilog language. We propose to add ABV as SystemC monitors on top of the original design. Doing so, an important goal will be achieved, namely a uni£ed language which brings together enhanced design and assertion features that deliver increased designer productivity and smarter veri£cation. In the same time, considering SystemVerilog's standard assertions will take advantage from the result of an industry-wide effort to extend the Verilog language to include enhanced modelling and veri£cation features.**

I. Introduction

SystemC [9] is among a group of design system level languages (SLL) proposed to raise the abstraction level for embedded system design and veri£cation [8]. It is expected to make a stronger effect in the areas of system architecture, co-design and integration of hardware and software [7].

The veri£cation of a SystemC design is a more serious bottleneck in the design cycle. Going further in complexity and considering hardware/software systems will be out of the range of the nowadays used simulation based techniques [6]. Classical veri£cation 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.

For instance, the main trends in de£ning new veri£cation methodologies are considering a hybrid combination of formal, semi-formal and simulation techniques. One of the widely used techniques is Assertion Based Veri£cation (ABV) [5]. In fact, assertions are said to be the next big breakthrough that will enable engineers to continue to design and verify larger and more complex designs. Assertion-based methodologies will bring much needed structure to the current set of ad-hoc testbench and monitoring techniques used by most designers for simulation, as well as enable more widespread adoption of emerging formal and semi-formal veri£cation technologies.

Accellera currently has three standardization efforts underway that will further enable assertion-based veri£cation: Formal Property Language—Sugar [4], SystemVerilog Assertions (SVA) [3] and Open Veri£cation Library (OVL) [1]. First, there is a formal property language effort, which is de£ning the Sugar language standard for property speci£cation which will provide new capability to express abstract, higher-level speci£cations of design functionality. Next there is the effort to add a native assertion construct and capability to SystemVerilog, which will provide a simpler and more expressive way of including assertions directly into Verilog designs. Finally, the OVL effort de£nes the standard library of assertions, providing a consistent method of reporting and controlling these assertions.

In this paper we propose to augment SystemC by the support of SVA. To do so, we consider the same syntax and semantics of SVAs. 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 veri£cation process (representing the code to verify the assertion itself). The design is updated in order to be correctly connected to the assertions' monitors by offering access to the signals involved in the assertion. During simulation, all assertions monitors act as a part of the design. In order to get advantages of the SystemC OO nature, every monitor is only triggered whenever at least one of its signals is updated, which leads to a more ef£cient simulation.

The rest of this paper is organized as follows: Section 2 introduces the SystemC library. Section 3 presents SystemVerilog Assertions. Section 4, de£nes the extension of SystemC by SVA. Finally, Section 5 concludes the paper.

II. SystemC Language

SystemC is a set of C++ class de£nitions and a methodology for using these classes [7]. SystemC is built on standard C++. The core language consists of an event-driven simulator as the base. It works with events and processes. The other core language elements consist of modules and ports for representing structures. Interfaces and channels are used to describe communications. The primitive channels are built-in channels that have wide use such as signals, semaphores and FIFOs. SystemC provides data types for hardware modelling and certain types of software programming as well.

Events occur at a given simulation time. Time starts at 0 and moves forward only. Time increments are based on the default time unit and the time resolution. Three main concepts are used here: initialization, elaboration and simulation semantics.

- *Initialization*: is the £rst step in the SystemC scheduler. Each process is executed once during initialization and each thread process is executed until a wait statement is encountered.
- *Elaboration*: is de£ned as the execution of the *sc_main()* function from its entry point to the £rst invocation of *sc_start()*.
- *Simulation*: The SystemC simulator controls the timing and the order of process execution, handles event noti£cations and manages updates to channels. It supports the notion of *deltacycles*, which consists of the execution of evaluate and update phases. The number of *deltacycles* for every simulation time depends on the simulation itself.

III. SystemVerilog Assertions

The SystemVerilog standard is the result of an industry-wide effort to extend the Verilog language in a consistent way to include enhanced modeling and veri£cation features. A key feature of SystemVerilog is the SystemVerilog Assertion (SVA), which uni£es simulation and formal veri£cation semantics to drive the design for veri£cation methodology.

The semantics of SVA are de£ned such that the evaluation of the assertions is guaranteed to be equivalent between simulation and formal veri£cation. This equivalence ensures that multiple tools will all interpret the behaviors speci£ed in SVA in the same way. Moreover, the uni£cation of assertions with the design and veri£cation code streamlines the interaction between the assertion and the testbenchto augment the power of assertions. In particular, SystemVerilog allows assertions to communicate information to the testbench and allows the testbench to react to the status of assertions without requiring a separate application programming interface (API).

SystemVerilog provides two types of assertions: *immediate* and *concurrent*.

A. Immediate Assertions

Immediate assertions are procedural statements that can occur anywhere within always or initial blocks, and include a conditional expression to be tested and a set of statements to be executed depending on the result of the expression evaluation. The syntax of an immediate assertion is:

$$immediate\_assert\_statement\ ::= \\ assert(expression)[[pass\_statement]else[fail\_statement]]$$

The expression is evaluated immediately when the statement is executed, exactly as it would be for an if statement. The pass_statement is executed if the expression evaluates to true, otherwise the fail_statement is executed.

B. Concurrent Assertions

The real power of SVA, both for simulation and formal veri£cation, is the ability to specify behavior over time, which VHDL assertions cannot do. For instance, it is possible today to use Property Speci£cation Language (PSL) [**?**] with VHDL in the form of pragmas, or structured comments, that begin with the string "-psl". But a better solution would be to add PSL declarations and statements to VHDL as £rst-class language constructs. In VHDL-200x [10], the Simple Subset of PSL will be integrated as part of VHDL.

Concurrent assertions provide the ability to specify such sequential behavior concisely and to evaluate that behavior at discrete points in time, usually clock ticks (such as "posedge clk"). The syntax of concurrent assertions is:

$$concurrent\_assert\_statement\ ::= \\ assert(sequential\_expr\_or\_prop)[[pass\_statement]\ else \\ [fail\_statement]]$$

The concepts and components that make up concurrent assertions can best be understood as a set of layers, each building on the layer as described in Figure 1.

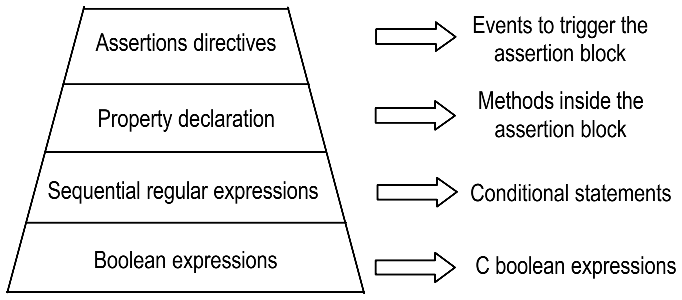

Fig. 1. Mapping between SystemVerilog assertions and SystemC objects.

The basic function of an assertion is to specify a set of behaviors that is expected to hold true for a given design or component. The Boolean expressions layer is the most basic one, and speci£es the values of elements at a particular point in time, while the sequential regular expressions layer builds on the Boolean layer to specify the temporal relationship between elements over a period of time. The property declarations layer builds on sequences to specify the actual behaviors of interest, and the assertion directives layer explicitly associates these behaviors with the design and guides veri£cation tools about how to use them.

To ensure consistency between simulation and formal veri£cation tools, which apply a cycle-based view of the design, concurrent assertions in SystemVerilog use sampled values of signals to evaluate expressions. The sampled value of signals is de£ned to be the value of the signal at the end (for instance, at read-only synchronization time as de£ned by the Programming Language Interface (PLI) [3]) of the last simulation time step before the clock occurs. This way, a predictable result can be obtained from the evaluation, regardless of the simulator's internal mechanism of ordering and evaluating events.

IV. Extending SystemC by SVA

To add SVA to SystemC two options are possible: integrate the SVA as part of the library or on top of the library. The £rst case presents a radical change of SystemC requiring adding new constructors to the library (*assert* for example). Besides, the SystemC simulator and semantics must be updated in order to manage and verify correctly the assertions. This choice may seem to be the most ef£cient as assertions will be de£ned in SystemC the same way they are integrated in SystemVerilog. Nevertheless, considering the OO aspect of SystemC and its modular structure, it is easier yet probably more ef£cient to add assertions on top of SystemC. In fact, any assertion can be seen as a monitor having as input some of the design signals, performing a veri£cation operation and giving as output a status ¤ag. The open question facing this latter approach is how to update the design in order to connect the assertions monitors.

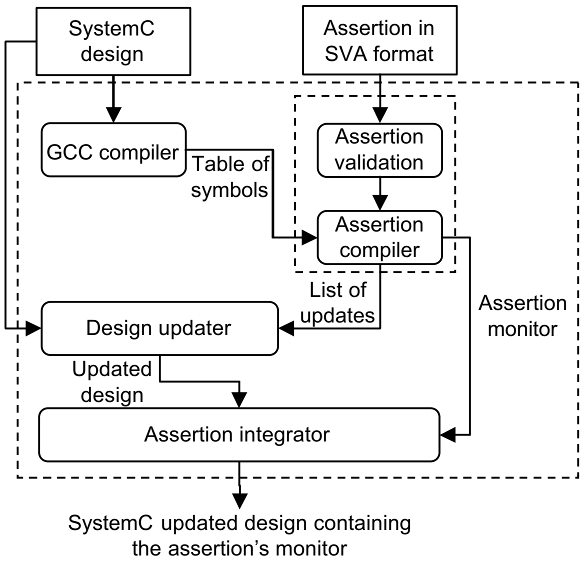

Fig. 2. Methodology of Extending SystemC by SVA.

Figure 2 shows the proposed methodology to construct and integrate SVA into SystemC design. We £rst start by collecting the information about the environment from the SystemC compiled code. To do so we consider the symbol £le generated from the Gnu-C-Compiler (GCC). This step is needed in order to localize which signal belongs to which module. Then, the assertion is validated and compiled. The validation phase veri£es 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 existent design, this latter needs also 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 and input to the design updater which performs the required modi£cations to the original SystemC design. These modi£cation will not affect the behavior of the design since they will only get some signals connected to the assertion monitor as *read-only*.

To assertion module is then connected to the updated design. This module will be instantiated in the main function of the SystemC design (*sc_main*) and connected to the appropriate existent modules. The resulting code when executed will therefore consider the assertion monitor as part of the design.

The assertion compiler generates the SystemC code that corresponds to the input assertion which includes: Boolean expressions, sequential expressions and properties. We will consider Boolean variables as SystemC signals (*sc_signal*) in order to get bene£t of the object nature of this module and to be able to integrate any variable as part of the monitor constructor section (containing the triggering conditions).

A. Sequential Expressions

SystemVerilog includes the ability to specify sequential expressions or sequences of Boolean expressions with temporal relationships between them. To determine a match of the sequence, the Boolean expressions are evaluated at each successive sample point, de£ned by a clock that gets associated with the sequence. If all expressions are true, then a match of the sequence occurs. The most basic sequential expression is something like *event1* followed by *event2* after three-clock cycles" which is represented in SystemVerilog syntax as: "*event1* # # 3 *event2*".

In this previous example, the "##3" indicates a three-clock delay between successive Boolean expressions in the sequence. Every sequence will be represented in SystemC as a list of members of the the assertion monitor. The clock cycles will be represented as counters. The code corresponding to the previous example is given by:

```
sc_in<bool> event1;
sc_in<bool> event2;
sc_in<int> counter = 0;
```

The *counter* variable is updated in the sequence validation method as follows:

```
if(event1)
  counter = 1;
if (counter > 0)
     counter++;
if(counter == 3) {
  if(event2)
  {
    counter = 0;
    return TRUE;
  }
  else
  {
    counter = 0;
    return ERROR;
  }
return Pending;
}
```

The main operations de£ned over sequences are summarized in Table I.

B. Property Declarations

The property layer allows for more general behaviors to be speci£ed. In particular, properties allow users to invert the sense of a sequence (e.g., when the sequence should not happen), disable the sequence evaluation, or specify that a

TABLE I
SVA SEQUENCE OPERATIONS.

Operation	Syntax	Meaning
Concatenation	seq1 ##1 seq2	seq2 begins on the clock after seq1 completes
Overlap	seq1 ##0 seq2	seq2 begins on the same clock seq1 completes
Ended Detection	seq1 ##1 seq2.ended	seq2 completes on the clock after seq1 completes (regardless when seq1 started)
Repetition	seq1 [*n:m]	repeat seq1 a minimum of n and maximum of m times.
First Match Detection	£rst_match(seq1)	if seq1 has multiple matches, consider the £rst one.
OR	seq1 or seq2	compound sequence that matches when seq1 or seq2 matches
End	seq1 and seq2	compound sequence that matches when both seq1 and seq2 match

sequence be implied by some other occurrence. The property construct allows these capabilities using the following syntax:

$property\_declaration$ ::=
$property\_name[formal\_item(\ , \ formal\_item)]$;
$assertion \ varaibles \ declaration$
$property\_specification$
$endproperty$

$property\_specfication$ ::=
$property\_name[formal\_item(\ , \ formal\_item)]$;
$[clocking\_events]$
$[disable$ *iff* $(expression)$] $[not]$ $property\_expr$

$property\_expr$ ::= $sequence\_expr \mid implication\_expr$

The important difference between sequences and properties resides in the fact that these latter are triggered by other signals other than clocks. As a result, the representation of a property in the assertion monitor will contain two parts:

- Property veri£cation method: A method that is responsible for the veri£cation of the assertion.
- Triggering conditions: a list of conditions that activates the veri£cation of the assertion (a signal update for example).

As an illustration consider the property: "as long as the *test* signal is low, check that the *abort_seq* sequence does not occur". This can be written in SVA as:

$property \ p1$;
@ $(posedge \ \ clk)$ $disable$ *iff* $(test)$ not $abort\_seq$
$endproperty$

This assertion is represented in the SystemC assertion monitor as method triggered when the signal *test* is low and performing the following code:

```
if(test.in()) {
  if(abort_seq)
    return ERROR;
  else
    return TRUE;
}
```

We note that in practice, most sequential assertions are expressed as some form of implication "when this happens, then that will happen", and thus require the assertion writer to specify the antecedent expression to trigger the assertion (for the previously given assertion the condition was *test.in()* set to TRUE). The object nature of the SystemC assertion monitor as a "*SystemC Module*" offers more ¤exibility to de£ne this kind of assertions.

V. CONCLUSION

In this paper we presented a methodology to integrate SystemVerilog Assertions (SVA) support to SystemC. We proposed to translate SVA into SystemC monitors connected to the design in order to verify some assertions during simulation. Our approach takes advantage from the object-oriented nature of the C++ language and the events concept of the SystemC library. We offered a hierarchical transformation of SVAs starting from basic Boolean expressions and getting to procedural assertions; therefore covering completely SystemVerilog assertions.

As future work, we consider to augment the assertion layer we de£ned for SystemC by: (1) A support for customizable messaging resulting from the assertions; (2) A communication between the assertions and the testbench and vice-versa; and (3) An optimization of the assertion code in order to get faster veri£cation time.

REFERENCES

[1] Accellera Organization. Open Veri£cation Library, Assertion Monitor Reference Manual, pp. 234, 2003.
[2] Accellera Organization. Property Speci£cation Language reference manual, Version 1.01, Accellera Organization, pp. 122, 2003.
[3] Accellera Organization. SystemVerilog 3.1 Accellera's Extensions to Verilog, Accellera Organization, pp. 374, 2003.
[4] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze and Y. Rodeh. "The temporal logic sugar". In Computer-Aided Veri£cation, volume 2102 of Lecture Notes in Computer Science, pp. 363-367, Springer-Verlag, 2001.
[5] B. Bentley. Validating the Intel Pentium 4 Microprocessor. In Proc. of the 38th Design Automation Conference, pp. 244-248, 2001.
[6] M. Kantrowitz and L. Noack. *"I'm Done Simulating: Now What? Veri£cation Coverage Analysis and Correctness Checking of the DECchip21164 Alpha Microprocessor"*. In Proc. ACM/IEEE Design Automation Conference, 1996.
[7] Open SystemC Initiative. SystemC 2.0.1 language reference manual. Open SystemC Initiative, pp. 428, 2003.
[8] P. R . Panda. SystemC: a modeling platform supporting multiple design abstractions. In *In Proc. of the 14th International Symposium on Systems synthesis*, pp. 75–80, 2001.
[9] SystemC: http://www.systemc.org, 2004.
[10] VHDL-200X (The Future of VHDL): http://www.eda.org/vhdl-200x, 2004.