PSL AND SVA: TWO STANDARD ASSERTION LANGUAGES ADDRESSING COMPLEMENTARY ENGINEERING NEEDS

John Havlicek, Freescale Semiconductor, Inc., Austin, TX
Yaron Wolfsthal, IBM Haifa Research Lab, Haifa, Israel

1 Introduction

On May 29, 2003, the Accellera EDA standards organization announced the official approval of Accellera PSL (Property Specification Language), based on the Sugar language from IBM, and of SystemVerilog, which contains an assertion capability known as SVA (SystemVerilog Assertions). SVA combines features from Synopsys OVA, Motorola CBV, and Accellera PSL. Since then, these assertion languages have been developed further within Accellera. Throughout this paper, "PSL" and "SVA" refer to the PSL 1.1 and SVA 3.1a versions of the languages, respectively.

A natural question that arises in this context is whether the semiconductor industry can benefit from two standards for assertion languages. More practically, for engineers who are about to embark on a new project, a pressing issue is that of whether to use one assertion language or the other.

In this paper, we aim to provide background data for engineers, engineering managers and EDA methodologists about the basic differences between PSL and SVA. The actual decision of which language to use in a specific system, microprocessor or ASIC projects depends on numerous factors, and describing that decision process is beyond the scope of this article. However, we believe that the key principles described herein will aid in the selection of the appropriate assertion language to use for the task at hand. Practically, we feel that in many cases engineers will want to have working knowledge of both languages. In fact, with the recent conclusion of the work done by a special committee chartered by Accellera to make sure the languages are made as close as possible, developing a working knowledge of the two languages is a straightforward task [1]. This alignment work was accomplished primarily in semantics, allowing tool builders to bridge between the languages. Engineers need only to recognize the syntax and precedence differences.

Before we move on, for completeness of exposition, a short overview of assertions and assertion-based verification (ABV) is in order. The interested reader is referred to the literature for additional breadth on the topic (cf., e.g. [2]). Generally speaking, ABV is a powerful paradigm for functional verification that augments and improves earlier approaches. As the complexity of hardware designs has grown to a degree that exposes limitations in the traditional approaches, the need for a better design methodology, one with improved levels of observability of the design behavior and controllability of the verification process, has become clear. ABV has been identified as a modern, powerful verification paradigm that can, if done right, assure enhanced productivity, higher design quality, and, ultimately, faster time to market and higher value to customers. With ABV, assertions are used to capture the required temporal behavior of the design in a formal and unambiguous way. The design can then be verified using dynamic and static verification technologies to assure that it indeed conforms to the design intent as captured by the assertions. One key characteristic of ABV is that the assertions capture the correct design behavior on a cycle-by-cycle basis and can accordingly be used to verify intermediate behaviors. As a result, assertions can detect an incorrect design behavior at the time and place it happens. This significantly improves the ability to find and fix bugs without relying on final simulation results and therefore serves to shorten turnaround time for the design process.

It should be clear from the above discussion that a suitable language should be available to capture the functional specification of the design, including assumptions, obligations, and invariants. PSL and SVA are two such languages.

A sample assertion is shown below in both PSL and SVA.

PSL:

```
assert always (
  {req && ack} |=>
  {!req within gnt[->1]}
)@(posedge clk);
```

SVA:

```
always @(posedge clk)
assert property (
  req && ack |=>
  (!req within gnt[->1])
);
```

This property reckons time according to the posedge of `clk`. When checked in simulation, the property says that if `req` and `ack` are both true at a time point, then, beginning at the next time point, `req` must be false up to and including the first time point at which `gnt` is true.

2 Why Two Languages?

As can be seen from the above examples, there is a high degree of similarity between PSL and SVA. This is, effectively, the result of the alignment work done by Accellera. Still, what are the differences between the languages, and why and where should each be used? The answer to this question requires an understanding of the different design decisions and objectives of the two languages, as well as their different language-theoretic foundations and purposes. To this end, the following section provides a user perspective of the differences, and the subsequent section clarifies the separate infrastructures of the languages.

3 User Perspective

SVA is part of and tightly tied into SystemVerilog. As a result, SVA can be written directly as a part of SystemVerilog designs and testbenches. SVA also inherits the expression language of SystemVerilog, including its data types, expression syntax, and semantics. PSL is a separate language specifically designed to work with many HDLs and their expression layers. As a result, PSL cannot be written directly as a part of any HDL. However, PSL properties can be attached to HDL models using binding directives, and tools can support PSL inclusion in HDLs via comment pragmas. Similarly, SVA cannot be written directly into HDLs other than SystemVerilog, but tool support for the use of SVA with other HDLs is possible through binding directives and comment pragmas.

3.1 User View of SVA

The full SystemVerilog language addresses needs of both hardware designers and verification engineers. Its features support the design and verification of hardware from the block level up to the system and full-chip levels. In addition to the assertion sub-language, SVA, these features include sophisticated software constructs for the design of complex SoCs and for the development of the verification testbenches to validate them.

There are several advantages to having SVA integrated with the full SystemVerilog language.

A designer can use SVA to embed assertions directly into the hardware design definition and/or into the testbench definition. These "white box" assertions record assumptions, expectations, and intentions of the designer that can quickly pinpoint design or integration mistakes and that are difficult to recapture after the design phase. The assertion representation is at a level of precision that is not easily rendered in natural language and that enhances the documentation of the design. Designers and verification engineers can also use SVA to define temporal correctness properties and coverage events external to the design code. A binding construct allows externally defined assertions to be attached to the appropriate signals in the design model or in a SystemVerilog testbench.

The tight coupling of SVA with the full SystemVerilog language means that assertions can be written to interact with other testbench components in powerful ways and without crossing the boundary of a programming language interface. For example, through the use of *action blocks*, the passing or failure of an assertion can be defined to trigger execution of a specific block of SystemVerilog code. The code in the action block might call a failure handling task, update a testbench coverage database, or influence the heuristic parameters of a reactive or self-adaptive testbench. As another example, an assertion can receive information by referring to

an auxiliary HDL model constructed as part of the SystemVerilog testbench.

SystemVerilog also provides a feature for attaching method calls to the detection of a temporal event within an assertion. The method calls can be passed any data in the local state of the assertion at the time the event is detected, thereby enabling the communication of fine-grained information about the event to other components of the SystemVerilog testbench. As a result, assertions can be a convenient and effective construct for the development of SystemVerilog testbench monitors.

In the future, SVA work will include investigations to provide adapters to other languages. Some people are already working on extending the binding construct of SystemVerilog to be able to access a VHDL instance.

3.2 User View of PSL

Practically, only some companies can adopt a single language approach. Most have to deal with both VHDL and Verilog For example, they may import IP from third parties who use the 'other' language, or as a result of acquisition there may be different divisions using different languages.

Furthermore, larger companies doing system-level design are often using, or planning to use, SystemC. These companies are looking for a way of writing assertions starting at the system level, with the expectations that such assertions can flow down to the RTL domain with little or no modification and that those assertions, together with more developed at the RT level, will work transparently in both VHDL and Verilog contexts. In support of this initiative, an implementation of PSL for use with SystemC has been demonstrated at DAC'04 [3].

Another domain addressed by PSL is that of system verification. A good number of system design houses (e.g. IBM, Intel, more) employ this pre-RTL methodology, where a high-level description of the system is modeled in an FSM form and verified against the architectural requirements [4]. PSL provides special support for this powerful verification methodology using the GDL (Generic Definition Language) flavor. A different application of PSL will be its extension to analog and mixed signal domains. A working group sponsored by the EU is presently pursuing the definition of such extensions to PSL [5]. Yet another pressing application is the verification of asynchronous designs, and work on extending PSL to support such design style is underway [6]. It is conceivable that more domains, applications and language flavors for PSL - which by design is flavor-extendible - will come up in the near future. One such creative application of PSL is its use for aerospace control applications [7]; an earlier one (where the base Sugar language was actually used) is for validation of railway interlock protocols [8].

PSL provides the capability to write assertions that range from system-level - in various kinds of systems - down to RT level. PSL has a structure of multiple abstraction layers and a rich set of operators that can be used at different levels of abstraction. The low-level layer of PSL, which governs the application domain, can be easily adjusted to many applications and design languages (e.g., the PSL Boolean layer - is suitable for reasoning about RTL designs - and has Verilog, VHDL, and GDL flavors). Moreover, the application layer can be even extended or replaced by a different layer to support new applications. In summary, PSL is a multi-purpose, multi-level, multi-flavor assertion language. In contrast, SVA is tightly connected to the SystemVerilog language.

4 Language-Theoretic Perspective

In this section, we compare and contrast PSL and SVA from a language-theoretic point of view. At a high level PSL is divided into the Foundation Language (FL) and the Optional Branching Extension (OBE). These are really separate sub-languages of PSL. A FL formula and an OBE formula cannot generally be combined into a single PSL formula. There is no analogous division in SVA, which is comparable as a whole to the FL sub-language of PSL.

4.1 Linear and Branching Semantics

PSL FL and SVA are linear temporal logics. This means that their formulas are interpreted over linear "traces" (i.e., "computation paths") in which each state has at most a single successor. Both languages are well suited to the dynamic or simulation-based ABV paradigm, in which assertions are checked over particular simulation traces of a design interacting with a testbench. Both languages can also be used for static verification, in which a single verification

computation can achieve the effect of checking an assertion over all possible linear traces. Most engineers in most applications will find the linear logics PSL FL and SVA sufficient for their purposes.

PSL provides additional support for advanced formal verification via the OBE. The OBE is a branching temporal logic very similar to CTL [9]. This means that an OBE formula is interpreted over "computation trees" in which a state can have multiple successors, as, e.g., in the case of a design interacting with a non-deterministic environment. Multiple successors can be treated either conjunctively or disjunctively, and the treatment can vary from one point in the formula to another. Thus, the OBE is well suited for expressing properties, such as freedom from deadlock, in which multiple successors need to be treated differently in different parts of the formula. OBE formulas generally cannot be meaningfully interpreted over simulation traces. Therefore, the checking of OBE formulas is typically limited to static techniques. A discussion of the applicability of branching semantics in formal verification can be found in [10].

SVA has no branching semantics features.

4.2 The Linear Logics of PSL and SVA

Here we compare PSL FL and SVA, ignoring "forall" quantification in the former and local variables in the latter.

Both PSL FL and SVA are built over sublanguages of regular expressions. The regular expressions are used to define finite linear temporal patterns. In PSL FL, the regular expressions are called *SEREs* ("Sequential Extended Regular Expressions"), while in SVA the regular expressions are called *sequences*.

PSL and SVA are highly similar at the level of regular expressions. PSL offers more derived operators ("syntactic sugar") than SVA and fewer restrictions on multiply-clocked regular expressions. For the typical user, though, either regular expression sublanguage will be entirely adequate.

Both languages provide for promotion of a regular expression to a strong formula, meaning that the temporal pattern described by the regular expression must be evidenced in the linear trace. PSL also offers promotion of a regular expression to a weak formula, which is not a feature of SVA (except as captured by weak finite-trace semantics). Both languages provide implication operators for predicating the checking of a formula on match of the pattern specified by a regular expression.

Above the level of regular expressions, the two languages differ more substantially. Both languages offer the Boolean operators for combining formulas. PSL offers the full range of temporal operators from LTL [11] as language constructs: weak and strong "until", "globally", "eventually", weak and strong "next-time". From this list of operators, SVA offers only "globally" in the form of the SystemVerilog "always" or the implicit "globally" of a concurrent assertion. However, SVA gives access to the weak LTL operators through user-defined recursive properties, as discussed in the next sub-section.

Broadly speaking, above the level of regular expressions PSL provides uniform access to both safety and liveness operators, while SVA is more oriented towards safety. In practice, safety properties tend to be much more common than liveness properties, and liveness checking is typically meaningful only with static verification techniques.

4.3 Weak Linear Temporal Operators and Recursive Properties

As indicated above, PSL FL includes all of the LTL operators as formula operators. SVA has none of these operators at the formula level, although it does have "globally" at the assertion level. However, SVA has a feature that allows the user to define parameterized properties that are equivalent to the LTL "globally" and weak "until" operators. This feature is the *recursive property*.

For example, to get the effect of the PSL FL formula

```
always p
```

a user of SVA can define the parameterized recursive property

```
property my_always(p1);
  p1 and
  (1'b1 |=> my_always(p1));
endproperty
```

and instantiate

```
my_always(p)
```

Similarly, to get the effect of the PSL FL formula

```
p until q
```

a user of SVA can define the parameterized recursive property

```
property my_until(p1, p2);
  p2 or
  (p1 and
  (1'b1 |=> my_until(p1,p2)));
endproperty
```

and instantiate

```
my_until(p,q)
```

It follows that all of the weak formulas of PSL FL can be rendered in a straightforward way in SVA using recursive properties. The rendering in SVA is somewhat less convenient than in FL because the recursive property definitions have to be written or imported from a library.

4.4 Manipulating Data in Assertions

It is a common problem when writing temporal assertions that data values that are observable at one time must be referenced at a later time when they are no longer directly observable. For example, the value of a signal that is valid in one stage of a protocol may be needed to define correctness of a later stage when the signal is no longer valid. The assertion may also need to compute some arithmetic combination of data that are valid at various times in order to define correctness.

One general solution to this problem is to create an auxiliary state machine to capture and manipulate the data as required. The assertion can then be written to reference the state machine at the appropriate times. Both PSL and SVA support this approach. In PSL, one can use the relevant modeling layer to define the auxiliary state machine, while in SVA the SystemVerilog HDL itself can be used. A disadvantage to this approach is that the state machine can be quite complicated and error prone. The closer the auxiliary state machine is to being a reference model for the design, the closer its complexity tends to approach that of the design itself.

Both PSL and SVA provide alternatives to the auxiliary state machine approach to capturing and manipulating data for use in assertions. In PSL, universal quantification (`forall`) can be applied at the top level of a formula. This allows the assertion writer to introduce "dummy" variables that can be used to capture data at one point and reference it later. For example, the PSL FL formula

```
forall v in boolean :
always (
  {a && (v == e1)} |=>          (1)
  {b[->1]} |-> (e2 == v)
)@(posedge clk)
```

reckons time according to the posedge of `clk` and says the following:

> Whenever `a` is true, the boolean value of `e1` at that time must equal the boolean value of `e2` at the next strictly subsequent time such that `b` is true.

The dummy variable `v` effectively samples the value of expression `e1` when a is true and holds this value until it is needed for the later comparison with `e2` when `b` is true.

The use of `forall` for manipulating data has limitations. The semantics is not very useful when the dummy variable appears only in the consequent of an implication. For example, the PSL FL formula

```
forall v in boolean :
always (
  {a} |=>
  {b[->1] : v == e1 ;          (2)
    b[->1] : e2 == v}!
)@(posedge clk)
```

reckons time according to the posedge of `clk` and says the following:

> If `a` is true, then 1) there are at least two future times at which `b` is true, 2) `e1` must be equal both to `0` and to `1` at the first of these times, and 3) `e2` must be equal both to `0` and to `1` at the second of these times.

The contradictory requirements on `e1` and `e2` are unlikely to be the intent of the assertion writer. A more useful intended meaning is the following:

If `a` is true, then there are at least two future times at which `b` is true, and the value of `e1` at the first of these times must equal the value of `e2` at the second of these times.

This meaning is represented by the PSL FL formula

```
forall v in boolean :
always (
  {a} |=>
  (
    {b[->2]}!                    (3)
    &&
    ({b[->1] : v == e1}
    |=> {b[->1] : e2 == v})
  )
)@(posedge clk)
```

SVA supports the manipulation of data in assertions with special assertion variables, called *local variables*. The local variables are declared as part of a regular expression or a formula. A local variable can be assigned a value at the end of a match of any regular sub-expression. The value stored in the local variable can then be referenced later in the assertion. For example, the FL formula (1) above can be rendered in SVA as

```
property p1;
  bit v ;
  (a, v = e1) |=>
  b[->1] |-> (e2 == v);
endproperty
always @(posedge clk)
  assert property (p1);
```

Assignment to and reference of a local variable in the consequent of an implication does not result in contradictory requirements as in FL formula (2) above. FL formula (3) above can be rendered in SVAas

```
property p3;
  bit v ;
  a |=>
  (b[->1], v = e1) ##1
  b[->1] ##0 (e2 == v);
endproperty
always @(posedge clk)
  assert property (p3);
```

5 Summary

Clearly, SVA and PSL are different languages, each with certain unique advantages and disadvantages. In view of the discussion above, engineers will likely want to have a working knowledge and to use both languages, sometimes within the same project. At a high level, the choice between the two may depend on interoperability and marketing decisions. On a deeper technical level, there are fundamental language-theoretic differences between the two that may influence the choice, depending on the verification requirements and methodology. Lastly, we note that observations similar to ours about the applicability of different languages in different contexts have been made by other authors (cf. [12]), and a multiple-language approach has been argued to be a practical engineering methodology.

6 Acknowledgements

The authors gratefully acknowledge the comments of Cindy Eisner, Erich Marschner, and several anonymous referees from Synopsys.

References

[1] Accellera FVTC Alignment Subcommittee Final Report, http://www.eda.org/vfv/docs/alignment_final_report.pdf, February 11, 2004

[2] H. Foster, A. Krolnik, D. Lacey, Assertion-Based Design, Kluwer Academic Publishers, 2nd edition, 2003.

[3] S. Swan, “Enabling PSL Assertions in SystemC”, PSL/Sugar Meeting, DAC 2004, http://www.pslsugar.org/papers/pm2_stuart_psl_sysc.pdf

[4] C. Eisner et al., “A Methodology for Formal Design of Hardware Control with Application to Cache Coherence Protocols”, http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/ps/methodology.ps

[5] Y. Wolfsthal, “EU-Sponsored Deployment of PSL”, PSL/Sugar Consortium Meeting, DATE’04, Februar 2004, http://www.pslsugar.org/psl_meeting.html

[6] J. Willis, “Breaking the EDA barrier in Async Design”, EE Times, http://www.eetimes.com/in_focus/embedded_systems/OEG20030606S0035

[7] M. Moulin, L. Gluhovsky, E. Bendersky, "Formal Verification of Maneuvering Target Tracking", Proceedings of the AIAA Guidance, Navigation and Control Conference, Austin, Texas, 2003.

[8] Using Symbolic Model Checking to Verify the Railway Stations of Hoorn-Kersenboogerd and Heerhugowaard, http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/ps/trains.ps

[9] E. Allen Emerson, E.M. Clarke, “Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons”, Science of Computer Programming, Vol. 3, pp. 241--266, 1982.

[10] Existential Properties: Sample Applications, Education Material, IBM Haifa Formal Methods Group, August 2001, http://www.haifa.il.ibm.com/projects/verification/sugar/examples.html

[11] A. Pnueli, “The Temporal Logic of Programs”, Technical Report CS97-14, Mathematics & Computer Science, Weizmann Institute of Science, 1997, http://wisdomarchive.wisdom.weizmann.ac.il:81/archive/00000150/

[12] B. Bailey, “Verification Languages and Where They Fit”, EDA Forum 2003, http://edacentrum.ims.uni-hannover.de/dateien/downloadables/dateien/mentor-seminar.pdf