Property Specification: The key to an Assertion-Based Verification Platform

Harry D. Foster
Chief Architect
Verplex Systems, Inc.
harry@verplex.com

Abstract

Assertion-based verification—that is, user specified properties and automatic property extraction combined with simulation and formal techniques—is likely to be the next revolution in hardware design verification. This paper explores a verification break-through prompted by multi-level specification and assertion verification techniques. The emerging Accellera formal property language, as well as the Open Verification Library standards and the important roles they will play in future assertion-based verification flows are discussed. Furthermore, automatic property extraction techniques are explored—and their important roles in validating semantic consistency in the context of an RTL signoff flow.

1 Introduction

A change is taking place in the way we design and verify our designs that will revolutionize the industry and result in the equivalent of a synthesis productivity breakthrough in verification. This change demands that we move from natural language forms of specification to forms that are mathematically precise and verifiable, and lend themselves to automation. Property specification is the key ingredient of this revolution, whose end result is improved verification through an *intelligent testbench*. An assertion-based verification platform is an integral part of an intelligent testbench, which consists of the following key components:

1. ***verifiable testplans*** through property specification (that is, functional coverage models),
2. ***hardware verification languages*** (HVLs) combined with property specification to raise the abstraction level of testbench generation
3. ***reactive coverage driven testbenches*** based on property specification (assertions and functional coverage),
4. ***automated block-level methodologies*** such as smart block-level simulation stimulus generation based on interface properties (i.e., constraints),
5. ***exhaustive formal verification*** and ***semi-exhaustive property checking*** techniques.

This paper discusses the important role of *property specification* and automatic property extraction techniques in the context of an assertion-based verification flow.

1.1 Standards

One organization that works to support improvements in verification methodologies is Accellera (see www.accellera.org). Their mission is to drive worldwide standards that enhance a language-based design automation process. Recently, the Accellera Formal Verification Technical Committee completed the 1.0 LRM, which defines its property specification language (PSL) [Accellera 2003]. This *declarative* property language supports *top-down* (that is, functional specification-driven) design methodologies. Declarative property languages are ideal for specifying architectural and global properties, as well as defining interface specification during block-level partitioning.

In addition to PSL, the Accellera Assertion Committee has developed a standard for specifying RTL implementation properties directly within the designer's HDL through the *Open Verification Library* (OVL) [Bening and Foster 2001] and the new *SystemVerilog* procedural assertion construct proposal [Foster, et al. 2002]. The OVL provides a template for expressing a broad class of assertions *structurally* within the designer's RTL, while the new assertion construct facilitates expression of assertions *procedurally* during RTL development. Both the OVL and the new assertion construct enable *bottom-up* (that is, white-box) verifiable implementation practices, which improve simulation-based methodologies while providing a seamless path to formal verification.

Combined, these powerful and expressive formal property languages enable engineers to:

- specify properties as assertions and constraints for formal analysis,

- specify functional coverage models to measure the quality of simulation and identify holes,
- specify pseudo-random constraint-driven simulation environments, similar to the research by Yuan, et al. [1999]; and Shinmizu and Dill [2002].

1.2 Monitor-based specification

While standardizing a property language, such as PSL, is integral to addressing increased verification complexity, it is not the entire solution. Equally important to this revolution in design verification is an effective methodology that unifies traditional and formal verification within an assertion-based verification framework. Recently, monitor-based methodologies have emerged as a technique for unifying traditional and formal verification (for example, *FoCs-Automatic Generation of Simulation Checkers from Formal Specification* [Abarbanel, et al. 2000]). Other approaches include creating a protocol bus-monitor that examines an agent's output signals (as the monitor's input) and then generates a Boolean *correct*$_i$ output signal, which is true when agent *i* is compliant to the specification (for example, *Monitor-Based Formal Specification of PCI* [Shinmizu, et al. 2000] and *A Specification Methodology by a Collection of Compact Properties as Applied to the Intel Itanium Processor Bus Protocol* [Shimizu, et al. 2001]).

An effective unifying methodology includes the Accellera Open Verification Library (OVL), which provides the systematic elements of the methodology. The OVL incorporates a consistent and systematic means of specifying RT-level implementation properties structurally through a common set of assertion monitors. The OVL monitors act like a template, which enables designers to express a broad class of assertions in a common, familiar RTL format. Furthermore, the OVL capitalizes on the various Accellera assertion techniques by unifying the PSL *declarative* form of property specification with the new SystemVerilog (and VHDL) *procedural* form of specification within the library. Finally, these monitors address assertion-based methodology considerations by encapsulating a unified and systematic method of reporting, that can be customized per project, and a common mechanism for enabling and disabling assertions during the verification process. The reporting and enable/disable features use a consistent process, which provides uniformity and predictability within an assertion-based methodology.

2 Property specification

Informally, a *property* is a general behavioral attribute used to characterize (that is, a collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behavior).

When discussing properties, it is generally easier to view their composition as three distinct layers:

- the *Boolean layer,* which is comprised of Boolean expressions (for example, Verilog or VHDL expressions)
- the *temporal layer,* which describes the relationship of Boolean expressions over time
- the *verification layer,* which describes how to use a property during verification

Defining (or partitioning) a property in terms of the abstract view (that is, layer structure) enables us to dissect and discuss various aspects of properties. However, it is actually quite simple to express design properties; the three-layer view is merely a way to explain.

A property's *Boolean layer* is comprised of Boolean expressions composed of variables within the design model. For example, if we state that "*signal* en1 and signal en2 are mutually exclusive" (that is, a *zero-or-one-hot* condition in which only one signal can be active high at a time), then the Boolean layer description representing this property could be expressed in Verilog as:

```
(!(en1 & en2)
```

Notice that we have not associated any time relationship to the statement: "*signal* en1 and signal en2 are mutually exclusive". In fact, the statement by itself is ambiguous. Is this statement *true* only at time 0 (as many formal tools infer), or is it *true* for all time?

To remove all time ambiguities, a property's *temporal layer* describes the Boolean expressions' relationships to each other over time. For example, if *signal* en1 and signal en2 are always mutually exclusive (that is, for all time), then a temporal operator could be added to the Boolean expression to state precisely when the Boolean expression must hold (that is, evaluate *true*). This could be written in PSL as follows:

always !(en1 & en2)

There are many temporal operators in PSL (including *always, never, next, eventually, until*), which permit us to reason about very complex temporal relationships that potentially involve multiple Boolean expressions. The Boolean layer combined with the temporal layer form the basis of the *property*.

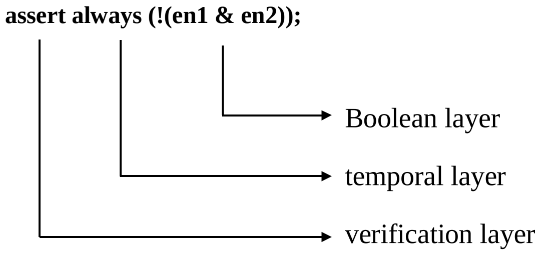
While a property's Boolean and temporal layers describe general behavior, they do not state how the property should be used during verification. In other words, should the property be asserted, and thus checked? Or should the property be assumed as a constraint? Or should the property be used to specify an event used to gather functional coverage information? Hence, the third layer of a property, which is the *verification layer*, states how the property is to be used.

Consider the following definitions for an assertion and a constraint.

- **assertion:** A given property that is expected to hold within a specific design. An *assert* keyword could be associated with the property (depending on the property language).
- **constraint:** A condition (usually on the input signals) which limits the set of behavior to be considered. A constraint may represent real requirements (e.g., clocking requirements) on the environment in which the design is used, or it may represent artificial limitations (e.g., mode settings) imposed in order to partition the verification task.

Look again at *signal* en1 and signal en2. We can specify this property as an assertion in PSL using the *assert* keyword as shown in Figure 1. This states that the property is to be treated as an assertion during verification.

Figure 1: Multiple layers of a PSL assertion



2.1 Declarative versus procedural specification

Assertions (or constraints) may be expressed either *declaratively* or *procedurally*. A declarative assertion is always active, and is evaluated *concurrently* with other components in the design. A procedural assertion, on the other hand, is a statement within procedural code, and is executed *sequentially* in its turn within the procedural description. Hence, declarative properties, such as PSL, are natural for specifying block-level interfaces, as well as other properties that must hold concurrently within a system. Similarly, a procedural assertion, such as the new SystemVerilog assert construct, is convenient for expressing algorithmic properties that must hold in the context (and sequential scoping) of procedural code.

For example, using the Accellera *declarative* formal property language PSL, the designer could express that the 8-bit bus *cntrl*[7:0] must possess the property of zero or one-hot as shown in Example 1, below.

Example 1: PSL declarative assertion

```
assert always ((cntrl & (cntrl – 1))==0) @(posedge (clk))
```

In Example 2, the zero or one hot assertion (using the new Accellera SystemVerilog *procedural* assert construct) is embedded directly within some procedural code.

Example 2: SystemVerilog procedural assertion

```
always (en or cntrl) begin
  if (en)
    assert @( posedge clk) ((cntrl & (cntrl – 1));
end
```

A key difference between the *declarative* assertion (shown in Example 1) and the *procedural* assertion (shown in Example 2) is that the declarative assertion concurrently monitors the assertion expression, while the procedural assertion only validates the assertion expression during sequential visits through the procedural code. If the 'en' signal is never TRUE in Example 2, then the procedural assertion will never be validated. This might be the intended behavior. However, care must be taken when assertions are deeply nested within **case** and **if** constructs in procedural code to prevent over constraining the assertion expression.

Alternatively, if the 'en' signal is a required condition on the assertion, then a declarative assertion must include this as part of the assertion expression. For example, we would require Example 1 to be re-written as shown in Example 3.

Example 3: PSL declarative assertion.

```
assert always (en -> ((cntrl & (cntrl – 1))==0))
       @(posedge clk);
```

To continue this progression, Example 4 demonstrates how to *structurally* express the same property shown in Example 3 using the Open Verification Library:

Example 4: OVL structural assertion.

```
   assert_zero_or_one_hot #(0,8) (clk, en, cntrl);
```

Example 5: FIFO Over and Underflow OVL Assertion

```
module fifo (clk, fifo_clr_n, fifo_reset_n, push, pop, data_in, data_out);
  parameter fifo_width = `FIFO_WIDTH;
  parameter fifo_depth = `FIFO_DEPTH;
  parameter fifo_cntr_w = `FIFO_CNTR_W;
  input clk, fifo_clr_n, fifo_reset_n, push, pop;
  input [fifo_width-1:0] data_in;
  output [fifo_width-1:0] data_out;
  wire [fifo_width-1:0] data_out;
  reg [fifo_width-1:0] fifo[fifo_depth-1:0];
  reg [fifo_cntr_w-1:0] cnt; // count items in FIFO
 . . . .
// RTL FIFO Code Here
 . . . .
 ‘ifdef ASSERT_ON
// OVL Assert that the FIFO cannot overflow
     assert_never no_overflow (clk,(fifo_reset_n & fifo_clr_n),
                      ({push,pop}==2'b10 && cnt==fifo_depth));

// OVL Assert that the FIFO cannot underflow
     assert_never no_underflow (clk,(fifo_reset_n & fifo_clr_n),
                      ({push,pop}==2'b01 && cnt==0));
 ‘endif
endmodule
```

2.2 Top-down versus bottom-up specification

The previous section introduced the work of Accellera that promotes improved top-down and bottom-up verification methodologies with the PSL declarative property language and the OVL and SystemVerilog procedural constructs. This section discusses a usage model for combining these multiple forms of specification.

In a specification-driven design and verification methodology, the design architect begins by creating a high-level specification for key characteristics of the design (such as communication protocols or complex arbitration schemes) using a declarative property language like PSL. This enables specification consistency-checking prior to design and implementation, and eliminates architectural bugs.

As the architect partitions the design (top down) into block-level components, prior to RTL implementation, block-level interfaces should be specified to form verifiable contracts between multiple block-level components. The Accellera PSL formal property language declarative form of specification is ideal for this type of specification. These formal interface specifications serve as constraint models during block-level formal analysis, as well as interface monitors during simulation. Furthermore, through this process, interface misconceptions between multiple engineers are resolved prior to RTL coding.

As the engineers begin RTL-development, they should take advantage of *assertions* and add this form of specification for any potential corner case concerns. For example, using the OVL assertions the designer can specify that a queue or FIFO will never *over* or *underflow* (see Example 5)—or that a set of signals is *one hot*—or that a specified expression (condition) will never occur. Engineers should embed these specifications directly within the RTL.

The declarative block-level interface specification (developed during top-down specification) combined with RTL implementation assertion (created during bottom-up RTL development) facilitates block-level formal analysis. Similarly, the designer will leverage the block-level interface specification in the future for pseudo-random constraint-driven simulation vector generation.

3.0 Automatic property extraction

Commercially available formal verification tools are emerging as a key advancement in the design verification revolution. These tools enhance verification methodologies by automatically extracting many design properties from the engineer’s HDL model. The tool then exhaustively verifies the properties using formal techniques. This enables the engineer to verify many design properties early in the design cycle without the need to create testbenches and test vectors. The following sections provide examples of properties that can be automatically extracted and verified by tools. This set of properties includes bus contention, bus floating, set/reset conflicts, RTL (Verilog) X-assignment “don’t care” checks, full or parallel case “don’t care” checks, and clock-domain crossing checks.

3.1 Semantic consistency property checking

Transformation verification, using formal combinatorial equivalence checking, has become mainstream for many design projects over the last few years. This process is a critical component of the design and verification flow that ensures *logical consistency* between transformed models within the flow. Transformation verification minimizes the need for gate-level simulation. However, when creating an RTL signoff methodology, the process of ensuring *semantic consistency* between the pre- and post-synthesis model must be addressed as well [Bening and Foster 2001].

The following definitions offer some insight into the significance of addressing logical and semantic consistency and X-state optimism.

Definition: Logical consistency
Referenced and revised design models are *logically consistent* if their combinational logic cones driving the next state latches and the output ports are functionally equivalent, modulo their don't-care space.

Definition: Semantic consistency
Referenced and revised design models are *semantically consistent* if, for all possible sequences of input vectors, the simulation results observed at all the latches and output ports are the same.

To illustrate the impact of semantic consistency, consider a particularly insidious situation in which design errors cannot be demonstrated during RTL simulation, yet are easily revealed using gate level simulation for logically equivalent circuits [Bening and Foster 2001]. This occurs because it is possible for an RTL and gate-level model to exhibit *logical consistency*, yet behave *semantically inconsistent*. That is, an equivalence checker proves that the two circuits are equivalent from a Boolean perspective. However, the pre- versus post-synthesis simulation results differ. To comprehend this phenomenon requires an understanding of RTL X-state optimism, which is explained below.

Definition: X-state optimism
Optimistically exercising the **default** (or **else**) branch, from multiple alternative branches, within a procedural **case** (or **if**) statement due to an X evaluation of the statement’s test expression.

The following Verilog code demonstrates RTL X-state optimistic behavior:

```
case (d)
  2’b01:   e = 2’b10;
  2’b10:   e = 2’b01;
  default: e = 2’b00;
endcase
```

If ‘*d*’ evaluates to *2’bXX*, the case statement will optimistically evaluate the default branch, assigning *e* the known value *2’b00*. Given an *XX* initialization state for *d*, then only one of the four possible branches during the start-up condition is tested, which can result in missing a functional bug.

The functional class of bugs hidden by RTL X-state optimism is generally the result of coding styles that yield semantic inconsistency, and include: (a) X assignment usage; (b) bit-vector select range overflow; and (c) synthesis pragmas, such as *full_case* and *parallel_case.* Semantic inconsistency problems for (c) are discussed in the following section.

3.2 Full case semantic problem

Example 6 illustrates a full case semantic problem. Examine the following RTL code, which contains a *full_case* synthesis pragma.

Example 6: RTL code for one-hot mux

```
module mux (a,b,S,q);
  output       q;
  input        a, b;
  input [1:0]  s;
  reg          q;

  always @(a or b or s)
  begin
   case (s) //synthesis full_case
      2’b01: q = a;
      2’b10: q = b;
   endcase
  end
endmodule
```

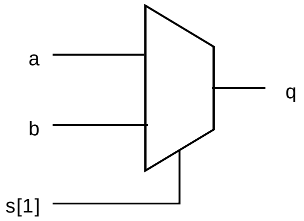


Figure 1: Synthesized gate for mux

Notice the RTL code in Example 6. If the *s* input variable ever assumes the unspecified values of *2’b00, 2’b11* or *2’bXX* (due to a functional bug in the logic driving s), then the *q* output variable will improperly behave as a latch during RTL simulation (that is, retain its previous, valid assignment). However, the synthesized logic in Figure 1 does not contain a latch.

For Example 6 and Figure 1, notice the case when *s==2'b11.* The circuits are semantically inconsistent since the RTL *q* output simulates to *a*, while the gate *q* output simulates to *b*. However, these two circuits will prove to be logically equivalent, since the *full_case* pragma instructs the equivalence checker (and synthesis tool) to treat any value of s other than *2'b01* or *2'b10* as a don't care.

It is possible to automatically extract properties derived from the designer's HDL code, and then apply formal techniques to validate the designer's intent. For example, in Figure 1 the property—*'s' will never equal 2'b00 and 's' will never equal 2'b11*—can be automatically extracted from the RTL code, and then formally verified to ensure semantic consistency.

Note that some designers might be tempted to assign the q variable in example 6 a default value of X. However, this can result in optimistic behavior further down stream in the RTL during simulation—which means that a bug might go undetected. Automatic property extraction is a better technique for validating the safety of all X assignment, since formal verification will not encounter X-state optimism.

3.3 Parallel case semantic problem

Even if the **case** statements are fully specified, it is still possible to encounter semantic inconsistencies due to the general use of synthesis pragmas within the RTL. Example 7 demonstrates a dangerous semantic inconsistency problem when a *parallel_case* pragma appears in the RTL. The problem occurs if there is a functional bug in the logic outside the module, that generates illegal values for the variables *a* and *b* (for example, *{a, b} == 4'b1111*). The RTL simulation always behaves as a priority encoder, while the synthesized gate-level model would encode both *y* and *z* in parallel (since synthesis uses the pragma to identify *don't care* values for optimization). Hence, the functional bug could be missed during RTL simulation since the RTL model optimistically behaves as a priority encoder, while the gate-level model behaves differently (as shown in Figure 2).

In Example 7, the designer's intent can be formally verified by automatically extracting the property—*it is not possible for both 'a' to equal 2'b11 and 'b' to equal 2'b11.*

Example 7: RTL Code

```
module encoder (y, z, a, b);
  output       y, z;
  input  [1:0] a, b;
  reg          y, z;

  always @(a or b) begin
    {y, z} = 2'b00;
    casez ({a, b}) //parallel_case
      4'b11??: z = 1'b1;
      4'b??11: y = 1'b1;
    endcase
  end
endmodule
```

Examples 6 and 7 demonstrated semantic problems that you may encounter through pragma use. In addition to these problems, general variable X assignments can exhibit optimistic behavior when evaluated as a test expression within a case or if statement. This can result in semantic inconsistency between the RTL and gate-level models.

Ensuring semantic consistency is a key component within an RTL signoff methodology. Hence, automatic extraction and formal analysis of RT-level design intent properties complements Boolean equivalence checking to ensure that both *logical consistency* and *semantic consistency* are preserved during design flow transformations—minimizing the designer's dependency on gate-level simulation.

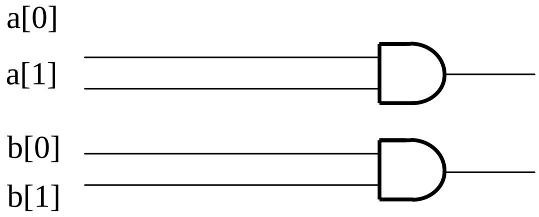


Figure 2: Synthesized Gates

5.0 Conclusion

In this paper we discussed the need for declarative, structural, and procedural mechanisms for expressing assertions. The emerging Accellera PSL, OVL, and SystemVerilog procedural assertions were introduced. Techniques for combining these languages were proposed, creating a top-down interface specification methodology with a bottom-up RTL implementation assertion methodology, which creates an effective assertion-based verification flow. This flow facilitates formal analysis, while improving traditional simulation methodologies. Finally, we demonstrated the important role automatic property extraction (and verification) plays within an RTL-signoff methodology.

Formal verification technology will become a key component within an assertion-based verification framework. This framework will enable the designer to verify a set of user defined properties (specified with the new Accellera PSL and OVL standards)—while supporting automatic property extraction and verification techniques.

References

[Abarbanel, et al. CAV 2000] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, Y. Wolfsthal, “FoCs: Automatic Generation of Simulation Checkers from Formal Specifications,” *Proceedings of the Computer-Aided Conference* (CAV), pp. 538-542, 2000.

[Accellera 2003] Accellera Property Specificatoin language http://www.eda.org/vfv/docs/psl_lrm-1.0.pdf

[Bening and Foster 2001] L. Bening, H. Foster, *Principles of Verifiable RTL Design*, Kluwer Academic Publishers, May 2001.

[Foster and Coelho 2001] H. Foster, C. Coelho, “Assertions Targeting A Diverse Set of Verification Tools,” *Proceedings of the 10-th Annual International HDL Conference*, March, 2001.

[Foster et al. 2002] H. Foster, P. Flake, T. Fitzpatrick, “Adding Design Assertions to SystemVerilog”, *Proceedings of the 11-th Annual International HDL Conference, March 2002*.

[Shinmizu et al. 2000] K. Shimizu, D. Dill, A. Hu, “Monitor-Based Formal Specification of PCI,” *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design,* pp. 335-353, November 2000.

[Shimizu et al. 2001] K. Shimizu, D. Dill, C-T Chou, “A Specification Methodology by a Collection of Compact Properties as Applied to the Intel Itanium Processor Bus Protocol,” In *CHARME’00*, Springer Verlag, pp. 340-354, 2001.

[Shimizu and Dill 2002] K. Shimizu, D. Dill, “Deriving a Simulation Input Generator and a Coverage Metric From a Formal Specification,” *Proceedings of the 39-th Design Automation Conference*, June, 2002.

[Yuan, et al. 1999] J. Yuan, K. Shultz, C. Pixley, H. Miller, A. Aziz, “Modeling Design Constraints and Biasing in Simulation Using BDDs, “ *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 584-589, November 1999