

Accellera Assertion Extension Requirements Proposal

Harry D. Foster
Hewlett-Packard
Richardson, TX
foster@rsn.hp.com

Abstract

This paper is intended as a strawman requirements proposal for assertion extensions to Verilog.

1. Introduction

This document is intended to form a strawman proposal for the Verilog assertion extensions. While establishing a set of requirements for the Verilog assertion extensions, the committee should consider the following goals:

- *Expressiveness*: The assertion extension should be expressive enough to cover most implementation properties likely to be specified by the design engineer.
- *Usability*: The assertion extension must be easy for the design engineer to understand and use.
- *Formalism*: The assertion language must have rigorous formal semantics to ensure correct compilation.

Assertion Extension Justification: The question arises whether assertion extensions are necessary. After all, many Hardware Verification Languages (HVLs) provide powerful language features that can be used to describe correct temporal behavior. These HVLs can be used for data generation and results analysis thru temporal specification. In addition, the Accellera Formal Verification committee is actively defining a property specification language that can be leveraged by multiple verification processes. Therefore, some will say either HVLs or formal Property Languages *should* be able to provide all the power necessary to express assertions within the design. However, a variety of stakeholders in the design and verification flow necessitate a variety of approaches to specifying design properties.

The reality of the contemporary design and verification flow requires a broader look at the issues, and an understanding of the goals of the various stakeholders. The value a white-box testing approach (through assertions) provides over a black-box approach has been validated by numerous sources. [Kantrowitz and Noack DAC 1996] [Taylor et. al. DAC 1998] [Bening and Foster Kluwer 2000] [Switzer et al. HDLCon 2000] [Foster and Coelho HDLCon 2001]. And, white-box testing can be performed with assertions, HVLs, or formal property languages. However, experience shows that verification engineers, whose goal is design validation, prefer an HVL approach to specifying properties of the design. On the other hand, the design engineers' focus is on implementation using hardware description languages (HDLs).

In addition to the dichotomy of goals, this issue also encompasses the areas of expertise of the various stakeholders. Quite often, the verification engineer lacks sufficient in-depth knowledge of design implementation details to provide effective white-box assertion coverage. Furthermore, during the course of RTL development, the design engineer may make low-level assumptions about the design's environment as well as other implementation assumptions. Experience has shown that if design assumptions or concerns are not captured during the process of RTL implementation, then many lower-level implementation properties are lost (that is, they will not be verified). For example, the *verification engineer* might wish to verify that a PCI bus controller exhibits correct behavior. This can be accomplished by using an HVL to generate correct bus functional stimulus and validate correct bus functional results. The verification engineer, however, would not validate specific implementation properties. Continuing with this example, assume that the PCI controller contains multiple embedded state machines. The *design engineer's* decision to implement a particular state machine as a one-hot versus some other type encoding is irrelevant to the verification engineer--provided that the PCI controller

exhibits correct bus functional behavior. Yet, capturing properties of the implementation (for example, one-hot) during the design process provides better white-box coverage.

Ultimately, the most effective overall approach for capturing *implementation* properties is to include assertions as part of the HDL during RTL development. Assertions directly encoded within the HDL, versus maintained separately through HVLs or property languages, simplify the integration of reused blocks (that is, design reuse). Experience has shown that difficult forms of assertion specification limit the number of assertions capture during the implementation process--thus poorer quality of white-box coverage. Furthermore, revisiting the HDL after the implementation phase to add in assertions also results in a poorer quality of white-box coverage.

Hence, although there is some overlap in HDL assertion specification, versus HVL and Property Language specification, the end user of the particular form of specification is different. Thus, convenience of HDL assertion specification must continue to be a consideration. This paper is intended to form a straw man proposal for assertion specification targeting the design engineer during Verilog implementation.

2. Definitions

2.1 Event

In this strawman proposal we define an *event* as a Verilog expression, which evaluates to a TRUE value at some point in time t during the course of verification. For example, if the expression $(c_ready == 1)$ evaluates TRUE at time t , then an *event* has occurred at time t in the verification environment.

2.2 Assertion

An *assertion* is a claim we make about an event or a sequence of events associated with the Verilog model. The assertion claims may be classified as either an *invariant* or a *liveness* property.

2.2.1 Invariant

An invariant assertion is a *static* event that must be valid for all time. There is no time relationship of events associated with an Invariant.

2.2.2 Liveness

A liveness assertion poses *temporal* behavior. In other words, there is a time relationship of events associated with it, whose correct sequence must be valid. For example, a temporal assertion can be viewed as an event-triggered window, bounding a specific property (that is, an event).

2.3 Assertion Expression

The assertion expression is an event represented by a Verilog expression.

2.4 Assertion Firing

The term firing in this proposal constitutes the condition when an assertion is violated.

3. VHDL Assertions

Details on the VHDL assertion statement are included in this proposal to provide the reader with an example of how other HDLs have implemented a simple assertion mechanism.

VHDL provides a language construct for specifying a *static invariant* assertion in procedural code. For example:

```
[label] assert event
      [report message]
      [severity level]
```

The VHDL assertion statement checks that a specified condition, or *event*, is true in a procedural fashion and reports an error if it is not.

The optional *report* clause specifies a message string for inclusion in error messages generated by the assertion. In the absence of a report clause for a given assertion, the string "Assertion violation" is the default value for the message string. The optional *severity* clause specifies a severity level associated with the assertion. In the absence of a severity clause for a given assertion, the default value of the severity level is ERROR.

Evaluation of an assertion statement consists of evaluation of the Boolean expression specifying the condition. If the expression results in the value FALSE, then an *assertion violation* is said to occur. When an assertion violation occurs, the report and severity clause expressions of the corresponding assertion, if preset, are evaluated. The specified message string and severity level (or corresponding default values, if not specified) are then used to construct an error message. The error message consists of at least:

- An indication that this message is from an assertion
- The value of the severity level
- The value of the message string
- The name of the design unit containing the assertion

To express *temporal* assertion or *liveness* properties requires constructing finite state machines to trap the temporal behavior within the VHDL code. The action performed due to a given severity level is determined by the tool.

4 Assertion Language Requirements

4.1 Syntax Language Features

4.1.1 Assertion Identifier

The Verilog Assertion Extension should have a unique identifier.

For example, if a primitive approach is adopted as the assertion specification mechanism (similar to existing gate primitives in Verilog), then an instance name similar to the gate level primitives could be used for identification. If an attribute approach is adopted as the assertion specification mechanism, a label identifier could be embedded for identification.

4.1.2 Assertion Reset

The Verilog Assertion Extension should have a mechanism for disabling the assertion.

This reset mechanism will disable the assertions from firing during verification and clear all states maintained by the assertion checker.

4.1.3 Assertion Sampling Clock

The Verilog Assertion Extension should have a mechanism for defining an optional sampling clock, whose [rising/falling] edge defines the appropriate time to evaluate the `assertion` expression.

4.1.4 Assertion Expressions

The Verilog Assertion Extension should support the entire synthesizable subset of Verilog expressions when representing an assertion expression.

NOTE: If not, we should explicitly state what is not supported.

4.1.5 Assertion Severity Level

The Verilog Assertion Extension should provide a mechanism for defining the assertion violation severity level.

The assertion committee might define various severity levels (for example, ERROR, WARNING, NOTE, and so forth).

4.1.6 Assertion Violation Action

The Verilog Assertion Extension should enable the user to define an optional action associated with an assertion violation in simulation.

One option for implementing this requirement is to allow the user to specify that the offending assertion call the *\$finish* task. Optionally, a user defined *\$PLI* call could be invoked upon an assertion firing. Possibly the user could specify the action as an option to the assertion construct? For example:

```
assert_type -id name -ck my_clock -rst rst_n -severity 0 -expr (a & b | c) -action $my_pli_call();
```

Again, I'm not promoting the syntax listed above. I'm just trying to emphasize that whatever assertion syntax we develop, it is desirable to associate either a *\$finish* or *\$PLI* call or action with a violation.

4.1.7 Concurrent and Procedural Assertions

The Verilog Assertion Extension should support both concurrent and procedural assertions.

A *concurrent assertion's* simulation semantics would be analogous to an instantiated primitive or instantiated module. In other words, the assertion would continuously monitor the *assertion expression* at every edge of a *sampling clock* as long as an assertion reset is inactive.

A *procedural assertion* is only activated when the particular line of code is visited in a procedural fashion (that is, the assertion expression is not continuously monitored). If procedural assertions are added as part of the language (and I think they should be seriously studied by the committee as a method of assertion specification), then clear simulation semantics must be defined. For example, we would need to define how the simulator would eliminate false firings due to simulation micro-time event execution. One possibility would be to associate (as an option) a sampling clock with a procedural assertion--then if a procedural assertion is activated, a call back occurs on the sampling clock to see if the assertion is still invalid.

Formal engines would obviously need to consider the nesting of *case* and *if* statements as part of the assertion expression. Experience has shown that users like procedural assertions (from my own experience at HP). Experience has also shown that users shoot themselves in the feet using procedural assertions due to over constraining the assertion expression by nesting in *if* and *case* statements (that is, they miss real bugs that would have been caught concurrently).

4.1.8 Event Rise or Fall Detection

The Verilog Assertion Extension should provide a convenient mechanism for specifying rising or falling events which can be used to create complex assertions.

For example, some kind of mechanism such as `-rise(expr)`, which is analogous to something like

```
always @(posedge ck)
    last_expr <= expr;
assign rise = (!last_expr & expr);
```

This feature has been requested by many designers to simplify coding assertion expressions. For example, currently designers detect the rising event as above and then, may use it in an assertion as follows:

```
assert_always test (ck, rst_n, rise ? test_expr : 1'b1);
```

This mechanism would be useful for temporal checks--such as validating the occurrence of multiple events. However, careful thought would be required prior to introducing any new edge detection mechanism into the language. Could we leverage what already exists?

4.1.9 Setting A Verilog Variable Upon Assertion Violation

The Verilog Assertion Extension should provide a convenient mechanism for setting a Verilog variable upon assertion violation.

NOTE: I do not favor this requirement since it potentially opens a can of worms. However, I think the committee should at least consider this requirement. This feature would enable the user to build very sophisticated assertions using the existing set of lower level assertions. Much of this would depend on language implementation. For example, this could easily be done with an assertion primitive approach--yet problematic using an assertion attribute approach.

4.1.10 Assertion Message Reporting

The Verilog Assertion Extension should provide a convenient mechanism for the user to specify a message string to be included in an error message generated by the assertion.

The committee might want to explore addition features associated with the assertion reporting mechanism. For example, the user might want an option to disable reporting, define their own reporting mechanism thru a \$PLI call, etc.

4.2 Semantic Language Features--Classes of Assertions

The following is a minimum set of recommended assertion *types* and their semantic meanings. This list is separated into *invariant* and *temporal* assertions. The list is derived from the HP user community and experience on common classes of assertions that the designer wished to specify. The committee should leverage the assertion experience from other assertion users to derive a comprehensive set (for example, customers of Verplex, 0-In, RealIntent, and so forth).

4.2.1 Invariance Assertions

assert always	Verilog assertion expression evaluates always TRUE on the rising edge of a sampling clock.	AG(expr)
assert implication	Verilog assertion expression is checked for implication	AG(X -> Y) Where X and Y are assertion expressions.
assert one hot	Verilog assertion expression is checked for one hot encoding. An option could be provided to permit all zero(one) or one_hot encoding.	AG((n & (n - 1)) == 0) Where n is the assertion expression
assert one cold	Verilog assertion expression is checked for one cold encoding. An option could be provided to permit all one(zero) or one cold encoding	AG((n & (n - 1)) == 0) Where n is the bitwise complement (~) of the assertion expression
assert parity	Verilog assertion expression is checked for even or odd parity	AG(^expr==PARITY) Where PARITY is either 0 or 1, and the expr is an assertion expression.
assert proposition	NOTE: This is similar to assert always; except, there is no sampling clock associated with it. This could be included as a sub-set requirement for assert always.	AG(expr)
assert no overflow	Ensure proper enqueueing and dequeuing behavior based on the size of the queue--where enqueue and dequeue are represented as assertion expressions (count enqueues and dequeues and validate range).	Formal definition goes here.
assert no underflow	Ensure proper enqueueing and dequeuing behavior based on the size of the queue--where enqueue and dequeue are represented as assertion expressions (count enqueues and dequeues and validate range).	Formal definition goes here.

4.2.2 Temporal Assertions

assert delta	Verilog assertion expression changes values within a specified DELTA range	$AG((\text{expr}=\text{X}) \rightarrow AX(\text{expr}=\text{X} \parallel \text{expr}=\text{X}+\text{DELTA}))$
assert next	Equivalent to the temporal logic AX^n operator: Given that proposition P1 is true in state s, the proposition P2 will be true at the <i>n-th</i> reachable states from x.	Equivalent to the AX operator.
assert event window change	Ensure that a Verilog expression changes values within an event-bounded window.	Formal definition goes here.
assert time window change	Ensure that a Verilog expression changes values with n number of clocks (could be combined with assert window change).	Formal definition goes here.
assert event window unchange	Ensure that a Verilog expression does not change values within an event-bounded window.	Formal definition goes here.
assert time window unchange	Ensure that a Verilog expression does not change values within a time-bounded window.	Formal definition goes here.
assert cycles	Ensure that a Verilog expression remains TRUE for a minimum/maximum number of sampling clocks.	Formal definition goes here.
assert frame	Ensure the proper number of cycles between two assertion expressions (for example, B will occur within MIN/MAX cycles after A).	Formal definition goes here.
assert transition	Ensure that a Verilog bit vector (for example, state) transcends to a legal set of values. Can be used to specify legal state transitions.	Formal definition goes here.
Assert no transition	Ensure that a Verilog bit vector (for example, state) will not transcend to a specified set of values. Can be used to specify illegal state transitions.	Formal definition goes here.

4.3 Assertion Usage Model

This section under construction--and will define how assertions are used in simulation and formal verification engines.