

Accellera Verilog++ Extension assertion construct Requirements

Contact
Harry D. Foster
harry@verplex.com

1. Introduction

This document specifies the IEEE-1364 Verilog++ extension assertion construct requirements. While establishing a set of requirements for the Verilog extension assertion constructs, the committee considered the following goals:

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

Extension assertion construct Justification: The question arises whether extension assertion constructs 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 necessitates 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 white-box testing (through assertions) provides over a black-box testing 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. On the other hand, during the course of RTL development, the design engineer makes 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 assertion 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 be a consideration. This paper is intended to form a strawman 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 possesses *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, i.e., 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 on how other HDL's 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 (i.e., event) is true in a procedural fashion and reports an error if it is not.

The optional report clause specifies a message string to be included 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 Language Features

4.1.1 Assertion Identifier

The Verilog extension assert construct should have a unique identifier associated with it. Assertion identifiers are important for error reporting and upkeep within multiple verification tools.

4.1.2 Assertion Reset

The Verilog extension assertion construct should have a mechanism for disabling the assertion. This reset mechanism will disable the assertions from firing during verification and clears all state maintained by the assertion checker.

4.1.3 Assertion Sampling Clock

The Verilog extension assertion construct 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 extension assertion construct expression should support the entire language of Verilog, and not be restricted to the synthesizable subset.

4.1.5 Assertion Severity Level

The Verilog extension assertion construct should provide a mechanism for defining the assertion violation severity level. The assertion committee might want to define various severity levels (e.g., ERROR, WARNING, NOTE, etc.). This could be accomplished using the Verilog attribute command.

4.1.6 Assertion Violation Action

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

For example, *\$finish* called when an assertion fires. Any valid Verilog statement or task should be permitted as an assertion violation action.

4.1.7 Concurrent and Procedural Assertions

The Verilog extension assertion construct 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's* is only activated when the particular line of code is visited in a procedural fashion (i.e., the assertion expression is not continuously monitored). A clear simulation semantics need to 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 unique sampling event or clock with each procedural assertion—then if a procedural assertion is activated, a call back occurs on the sampling event or clock to see if the assertion is still invalid.

4.1.8 Event Rise or Fall Detection

The Verilog extension assertion construct should provide a convenient mechanism for specifying rising or falling events.

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);
```

4.1.9 Options or Flags

The Verilog extension assertion construct should provide a convenient mechanism for enabling tools specific options, such as defining an assertion as a constraint to a formal tool.

NOTE: This might not be a necessary requirement. The new Verilog 2001 attribute construct might provide sufficient functionality to fulfill this requirement.

4.2 Language Constructs and Semantic Features--Classes of Assertions

The Verilog extension must support (a) *a mechanism for specifying procedural assertion*, and (b) *a mechanism for specifying concurrent assertions*.

(a) The simple procedural assertion, as discussed in section 4.1.7, would be analogous to the VHDL assertion construct, with the extension of associating a sampling clock for callback validation.

```
if (q_address `MAX_QUEUE) begin
    assert @(posedge ck) // <= Only for example, not promoting syntax
    : // Need some mechanism to associate a clock with assertion
```

(b) The simple concurrent assertion, which could be implemented similar to the existing Verilog gate primitive. In other words, the concurrent assertion would validate that an assertion expression evaluates to TRUE on every raising edge of a sampling clock.

```
// assertion primitive
assertion . q_overflow (ck, reset_n, (q_address > `MAX_QUEUE), . . . );
```

With these two simple assertions constructs, assertions that are more complicated can be constructed, via libraries similar to the OVL or vendor proprietary languages. In other words, all proprietary languages could be synthesized into a set of FSMs and the two basic assertion constructs. (A RISC approach to building an assertion language). This allows the maximum flexibility in extension assertion constructs.

4.3 Usage Model

Assertions techniques should be applied during the following steps of design:

4.3.1 Specification Design Refinement

As a first step, an HDL assertion methodology must support a specification based top-down refinement process. In other words, prior to coding RTL, all interfaces between blocks (or between multiple engineers at a sub-block level) should be specified in a verifiable format (for example, a bus functional model combining FSMs and assertions). These interface assertions form verifiable contracts between design partitions, and are the key to successful integration of formal and semi-formal techniques at the block level of the design.

4.3.2 RTL Implementation

The next step involves coding assertions during the RTL implementation phase. Assertions directly coded within the HDL, versus maintained separately, simplify integration of reused blocks. Embedding the assertions directly in the RTL facilitates:

- linting (in a single step) the RTL source for syntactical errors related to assertion specification; and
- capturing design assumptions and knowledge at the point of development; which becomes a permanent record of the design intent, along with the RTL.

While verification has historically been delayed until completion of design testbench, coding assertions at the RTL implementation phase enhances [insert something about reliability or confidence]. Rather than delay verification until the point in time when multiple blocks can be integrated into a large model, block-level formal verification can begin prior to the completion of testbenches (or test vectors) using an assertion methodology. This important step enables the designer to identify lower-level implementation bugs, while exploring missing interface assertions (for example, constraints).

4.3.3 Design Reviews

Designer peer review is an important step in the development process. This enables the engineer to identify design misconceptions and receive peer feedback on corner case concerns. The quality of existing assertions should be reviewed during this step. In addition, missing assertions should be identified during the review process to ensure high-quality, white-box coverage.

4.3.3 Assertions and Simulation

During the simulation verification step, experience demonstrates that assertions coding within the HDL dramatically reduces the engineer's debug effort. Bugs identified in the course of simulation through means other than assertion monitors indicate that the designer should add a new assertion that will trap this bug. This process addresses the following objectives:

- to capture the known corner case and document a permanent characteristic of the design;
- to provide the engineer with an increased awareness of how to code assertions; and
- to ensure (given the correct stimulus) that the bug can be identified in the original design, while providing a target to validate the modified design.

4.3.4 Assertions and Semi-Formal and Formal Verification

As the simulation environment stabilizes, semi-formal and formal techniques should be re-introduced into the verification flow to address two objectives. First, interface assertions provide constraints to the formal search engines during this step. Second, hierarchical formal verification can be achieved during this step by applying assume/guarantee techniques that utilize block-level interface assertions.

4.3.5 Design Reuse

Verification is a central component of design reuse. Assertions embedded in the HDL reusable blocks validate proper design reuse integration. Furthermore, these assertions form a verifiable specification.

4.3.6 Functional Coverage Models

Capturing implementation level “function coverage” points is an important step in ensuring complete implementation validation. Assertions (with lower severity levels) are a mechanism that can be used by design engineer to specify important coverage points for verification. Having a similar technique for specifying design errors (assertions) and lower-level functional coverage points is desirable.