Recent statistics suggest that Functional Verification is one of the primary bottlenecks in a typical design cycle. To counter this problem, the late 1990s saw a flurry of proprietary verification languages (a.k.a. HVLs), both procedural and declarative in nature. Accellera’s SystemVerilog was developed to address this very same issue – providing in 2005 a STANDARD and powerful language that combines the best of Design and Verification capabilities in a single language. The updated version of SystemVerilog will be published in 2009 as an IEEE standard. We anticipate that tool vendors will support both versions (the Accellera 2005, and the 2009 IEEE) with a compilation switch to allow users to select the desired version. SystemVerilog includes a powerful declarative subset that provides a means to formally capture temporal requirements of digital systems, more popularly known as SystemVerilog Assertions (or SVA). As with any new language, methodology, technology – the following questions come up immediately to a user’s mind:

- Is the use of SystemVerilog Assertions a good verification strategy?
- Are there frameworks that support SystemVerilog?
- Should SystemVerilog Assertions be used in the definition and verification of designs?

---
• Can/should the entire functional verification task be performed using Assertion-Based Verification (ABV)?
• What other verification strategies can be used with SystemVerilog Assertions?
• Are SystemVerilog Assertions solely restricted to applications that use SystemVerilog?
• How hard will it be to learn a new language?14
• Why should I describe the same thing in two different ways (e.g., RTL and higher perspective with assertions)?

This chapter addresses these questions. It assumes that the reader has a basic understanding of HDL and SystemVerilog. This chapter also provides some background information on Assertion-Based Verification and the history of SystemVerilog Assertions.

1.1 Design Verification Methodologies

Methodology is defined as a body of practices, procedures, and rules used by those who work in a discipline or engage in an inquiry. SystemVerilog Assertions (SVA) are an integral part of SystemVerilog and support Assertion-Based Verification (ABV) methodologies. It was developed to provide system engineers, designers, and verification engineers the means to describe complex behaviors about the design requirements and properties in a clear and concise manner. With the assertions unified syntactically with SystemVerilog, the user is able to define assertions either as externally defined design units (e.g., modules or checkers) bound to the Design Under Verification (DUV), or as embedded assertions inline with the RTL design and other verification code. These methods allow the tools to infer a great deal of information from the context of the surrounding code. Another useful feature of SystemVerilog Assertions is its testbench interaction, which unifies assertions with the design and verification language by communicating information to the testbench.15

Assertion-Based Verification is changing the traditional design process because this methodology helps to formally characterize the design intent and expected operations. Assertion-Based Verification also guides the verification task, quickens the verification because it provides feedback at the white-box level, and eases the design of the testbench as it transfers more of the "verifier" task onto the assertions, instead of user-defined finite state machine (FSM) models. In addition, SystemVerilog Assertions allow the verification environment to be reactive; meaning that based on results of the assertions (pass or fail), different actions may be taken. Formal verification can be used to start verification of a design before any test bench is available, thus speeding up the verification process and increasing the quality of the design. In summary, Assertion-Based Verification provides the following benefits:

• It's a verification technique that can be applied in simulation and formal verification to verify the behavior of a DUT.
• Instruments requirements with assertions
  o Clarifies requirements with executable language
  o Document assumptions on the behavior of inputs to a DUT
  o Enables tools to preview assertion waveforms
• Instruments design with assertions
  o Provides added visibility
  o Supports white-box testing into its internal state
  o Provides for functional coverage information
• Supported by IEEE Standards

14 That’s the purpose of this book, as we use complete examples and define the rules.
15 http://www.eedesign.com/features/exclusive/OEG20031015S0049
The concept of using assertions is not new. For ages, parents have told their children statements like “there are ten cookies in the cookie jar, and during the day, if you’re hungry, you must at most take ONE cookie! At the end of the day I want to see at least 9 cookies in the jar … and I’ll count the number of cookies”. That parenting statement is really an assertion that is verified by inspection. It can be expressed in SystemVerilog Assertions as follows:

```verilog
property pGetAcookie;
    eat_1_cookie iff ( hungry && cookie_count > 9)
endproperty : pGetAcookie
apGetAcookie : assert #0 (pGetAcookie);
```

This property states that I eat_1_cookie if and only if I am hungry and the cookie_count is greater than 9. The property is true when all three conditions are true, or when the signal eat_1_cookie is false and the expression (hungry and cookie_count>=9) is false. Otherwise the property will fail. The property is asserted using a deferred assertion so it is checked every timestep after all glitches have settled.

So, what is a property? What is an assertion?

**Property**: A property is a collection of logical and temporal relationships that represent a set of behavior, rules, or characteristics about a design. Properties can also be used to describe coverage targets or assumptions about the design environment. However, properties by themselves provide no verification or coverage requirements to be performed by tools (i.e., simulation or formal verification). To check for the correctness of a design behavior or for coverage an assertion or coverage statement (also called directive) is needed.

**Assertion**: An assertion is a statement that a given property is required to hold. It can also be a statement for the verification tools to verify that the property holds for the design under consideration. Note that properties can be named and assertions can be built from named properties. Assertions can have labels to allow tracking by the tools.

**Functional coverage**: It is a technique for ensuring that a sequence or a property occurs at least once. Functional coverage uses the cover, cover sequence, or cover property statements, and is unlike the assert or assert property statement (which ensures that a property always holds).

Over the years, as technology progressed and densities of designs increased, new verification methods were needed and developed. The 1970’s TTL and MSI technology appeared. Designs were defined with schematics on velum paper, and later electronically. However, verification was mainly performed with detailed paper design reviews, and through breadboard modules, which often used wire-wrap boards. During the 80’s, simulation tools became popular, but each vendor maintained their graphical design entry and proprietary simulation environment. Testbenches were hard to build with non-standardized tools, and verification relied on manually viewing the simulation results, with regression tests performed by comparing captured simulation runs against new updates. Any timing change would invalidate the golden copy.

In 1987, VHDL became a standard, but the use of HDLs was viewed critically because it was a major change to the “working methodologies.” Some companies restricted the use of HDL to the

---

16 Component instantiation was performed with the pasting of transparent self-adhesive pre-printed schematics of devices (e.g., gates, flip-flop, multiplexer) … that WAS technology!
definitions of combinational logic, which were then instantiated into the graphical schematic diagrams as modules. It took a paradigm shift, primarily caused by the acceptance of synthesis tools, to finally accept HDL as a design entry and verification vehicle. By then, government contractors started to impose HDLs as deliverables. In the 90’s it became evident to designers, except “old-timers” who still viewed schematics as a better methodology, that the days of schematic capture being the primary method of design entry was passé.

Static verification tools also started to emerge around that time to help in the static (i.e., no simulation) analysis of the designs for compatibility to design rules. As design densities increased, the use of more intelligent testbenches became the in-vogue paradigm. Self-checking testbenches with directed testing was then followed by pseudo-random, constrained-random stimulus generation, and finally transaction-based models. Implementation of self-checking testbenches, along with the introduction of better tools, such as coverage and better and faster simulators propelled the popularity of simulation. However, as designs became even more complex, the verification effort became more difficult with the current HDLs, particularly with Verilog because the language is very limited in data typing, and cannot execute reentrant functions and tasks. In addition, Verilog lacked the structures (e.g., classes) needed for the design of frameworks, such as VMM and OVM. 17

To overcome these deficiencies, hardware verification languages (HVLs) were introduced. These HVLs were tailored for simulation and integration with HDLs. HVLs introduced the concept of functional coverage and helped in tracking the verification progress. One class of HVLs (a.k.a. Assertion-languages) also introduced the concept of assertion-based verification (ABV), where design properties could easily be specified and verified. Assertion languages were also integrated with static (a.k.a. formal) verification tools to formally and mathematically prove or disprove the assertions. Formal tools exhaustively explore all possible sequences of input values in an attempt to prove that the assertion is valid. A counterexample for the failing assertion can be demonstrated in simulation from a sequence generated by a formal tool. Essentially, the formal tool creates a mathematical model of the design and the requirements, which includes a model of transitions for the system. A set of vectors could be generated of this transition model for a failing assertion.

With the emergence of new object-oriented languages, like Vera and SystemVerilog, newer verification methodologies based on frameworks recently became popular. These methodologies include VMM (Verification Methodology Manual) and OVM (Open Verification Methodology). These are not addressed in this book, as they are specialized methodologies addressed in many books. 18

17 Some of those limitations were removed in Verilog 2001. SystemVerilog 3.1 is Accellera’s Extensions to Verilog ®, and represents a set of extensions to the IEEE 1364-2001 Verilog Hardware Description Language to aid in the creation and verification of abstract architectural level models
18 Verification Methodology Manual for SystemVerilog, Janick Bergeron
A Pragmatic Approach to VMM Adoption, Cohen, Srinivasan, Kumari
Step-by-step Functional Verification with SystemVerilog and OVM, Sasan Iman
Writing Testbenches using SystemVerilog, Janick Bergeron
1.1.1 SystemVerilog Assertions in Verification Strategy

SystemVerilog’05 (Accelera version) and the IEEE 1800-2009 version has greatly extended the capabilities of Verilog’01 to create a unified language to meet both design and verification requirements of today’s complex designs. Among these various features are “assertions”. SystemVerilog Assertions are part of SystemVerilog. They enable the definition and verification of design properties using a concise, easy to read, and executable syntax. SystemVerilog Assertions with temporal sequences can express in just a few lines a complex FSM, whereas equivalent HDL code requires a much more complex model that is hard to follow. Because assertions are executable, they can be verified through simulation and/or static verification. Thus, assertions are essential because they:

- Describe complex temporal behaviors with small declarative statements
- Can be grouped into a checker construct, and inserted inline with RTL or other code
- Localize failures close to the origin
- Capture assumptions about the design and verification environment
- Can be represented with formal semantics
- Provide clear distinction between design and verification code
- Describe functional coverage points
- Can describe requirement specifications
- Can be represented with formal semantics
- Provide clear distinction between design and verification code
- Describe functional coverage points
- Can describe requirement specifications
- Significantly reduce the time to market because of early error detection

1.1.2 Are Assertions Independent from SystemVerilog Structures?

Assertions define properties that the design must meet. Many properties can be expressed strictly from variables available in the design. For example, in the typical handshake example with a request and an acknowledge, the following property and assertion can be expressed:

```verilog
property pReqAck;
// If request, then acknowledge should be asserted within 1 to 4 cycles.
// If no request, then property succeeds vacuously (See Appendix B for definition of vacuity)
$rose(request) |-> [#[1:4]] acknowledge;
endproperty : pReqAck
apReqAck : assert property (@ (posedge clk) pReqAck);
```

While properties are extremely powerful to capture control-centric design requirements, real designs include a mix of control and data elements. Hence a closer interaction between a higher level procedural language (with abstract data types, queues, sparse arrays etc.) and a declarative, formal language is needed to verify the data aspects of the design. The unified SystemVerilog standard supports such features. For example, queues and associative arrays can be used with SystemVerilog code as scoreboards, and they can also be used in the context of properties (though with certain restrictions). This concept is demonstrated in Section 6.3 for the definition of a synchronous FIFO. It is also demonstrated in Section 9.14 for the verification of data integrity in a memory module. Note that SystemVerilog Assertions allow for the definition of local variables for the storage of information, which can then be used at later cycles for the verification of properties. This capability is demonstrated throughout this book. Care must be taken when modeling data centric requirements via assertions as they may yield suboptimal results, such as with formal tools. It is recommended to use the right technology/feature for a given problem. SystemVerilog is a very rich language that should be used appropriately to achieve the best results. Thus, in some circumstances, it may be more appropriate to use assertions, whereas in other cases, coding the verification model with other SystemVerilog constructs is more efficient.
1.1.3 Are Assertions Useful for the Definition and Verification of Designs?

SystemVerilog Assertions is an integral component of SystemVerilog to support an Assertion-Based Verification methodology. Assertion-Based Verification is a verification technique that involves instrumenting the design with assertions to provide added visibility and white-box testing into its internal state.\(^{19}\) White-box functional verification using assertions enables the detection of bugs much earlier, at or near their source, rather than at outputs.

Assertions document the combinational and temporal (i.e., over cycles) properties of the design that must hold true (or never occur) for the design to be correct. They may be embedded into the design source text, or provided in separate files. Assertions can be checked either dynamically during simulation and/or statically through formal methods.

Assertions express functional design intent and can be used to express assumed input behavior, expected output behavior, interesting scenarios (for functional coverage), and forbidden behavior. For example, if a processor has a `read` and `write` interface signal, a designer might assume that “the `read` signal and the `write` signal should never be both active at the same time.” With assertion-based verification (ABV), that assumption can be captured and verified. This represents an example of an interface assumption. Interface assertions are used to check the interface protocol between blocks. Other types of assertions include application assertions and structural assertions, as shown in Figure 1.1.3. An application assertion represents a subsystem type of assertion, as viewed at a higher level of abstraction. Structural assertions address the internal verification of a subblock of RTL logic.

The use of ABV methodology along with SystemVerilog Assertions as the specification language provides the benefits outlined below.

![Figure 1.1.3 Types of Assertions in a Design](image)

\(^{19}\) Though in principle, assertions can be used to capture end-to-end behavior of the system, typical usage as of today is limited to lower-level implementation details.
1.1.3.1 Captures Design Intent

Often, very basic assumptions made by the designer are not explicitly captured as part of the HDL model. For example, a model may incorrectly always make a retransmission request upon a bus error. Yet the requirements for this design are that a retransmission request is only made in the SECURITY mode, and no retransmission should be made in the FAST mode, as bus errors can be tolerated in that mode. The following assertion not only captures the design intentions about the retransmission, but also clarifies the need for the abort command.

```verilog
property p_retransmit; // See Section 2.4.5.7 for the iff operator
    (mode==SECURITY && bus_err) iff (#1 abort_command #1 retransmit_cmd);
endproperty
```

```verilog
ap_retransmit : assert property(@ (posedge clk) p_retransmit);
```

This can also be expressed in SV2005 as follows:

```verilog
property p_retransmit1;
    (mode==SECURITY && bus_err) |=> abort_command #1 retransmit_cmd;
endproperty
```

```verilog
ap_retransmit1 : assert property(@ (posedge clk) p_retransmit1);
```

```
property p_no_retransmit;
    (mode==FAST && bus_err) |=> !abort_command #1 retransmit_cmd;
endproperty
```

```verilog
ap_no_retransmit : assert property(@ (posedge clk) p_no_retransmit);
```

1.1.3.2 Allows Protocols to be Defined and Verified

Temporal sequences can be used to specify and verify interface and bus protocols. Such assertions define not only the values of signals, but also the relationship between signals over time. Verification can be achieved through simulation and/or formal verification. An example of a protocol is a bus transfer that is completed (i.e., accepted OK) with a good CRC, and is failed when a CRC error is detected. The `crc_pass` or `crc_fail` occur within 256 clock cycles from when `ack` occurs.

```verilog
property p_bus_xfr; // Using a named property
    sync_accept_on(crc_pass) sync_reject_on(crc_err) // See Section 2.4.5.14
    $rose (req) |=> ack #1:256] done;
endproperty
```

```verilog
ap_bus_xfr : assert property(@ (posedge clk) p_bus_xfr);
```

Using SV2005, the above can be expressed as:

```verilog
ap_crc : assert property(@ (posedge clk)
    $rose(req) $rose(ack) #1:256] ((crc_pass || done) && !crc_err));
```
1.1.3.3 Reduces the Time to Market

An Assertion-Based Verification methodology reduces the time to market for the following reasons:

- ABV provides early detection of errors, especially when formal verification is used to start proving the properties prior to when a testbench is available.
- ABV facilitates debugging of failing tests as assertions can behave as monitors close to the source of error.
- Assertions can be grouped into a checker that can be inserted as a group, inline with code (see Chapter 5).

This reduction in time to market is demonstrated in Figure 1.1.3.3, and was proven on many projects.

1.1.3.4 Greatly Simplifies the Usage of Reusable IP

If an IP block contains assertions that express its input constraints, application configurations, and interface protocols, these assertions can check whether the block is used properly in a much larger system. Assertions added to IP blocks help to ensure proper integration of the IP. Typically the verification of SoCs, built using these IP blocks, involves the verification of interfaces rather than the individual block’s functionality. Hence an Assertion-Based Verification methodology fits the best, as compared to traditional verification methodologies. The common experience with IP-based designs has revealed that the majority of bugs adhere to the improper usage/configuration of an IP, rather than the functionality of the IP itself.

---

20 By permission from An Introduction To Property Checkers For Functional Verification, Andrew Jones and Jeremy Sonander, Saros Technology, http://www.averant.com
1.1.3.5 Facilitates Functional Coverage Metrics

Functional coverage is a technique for measuring how much of the functionality was exercised by verification. Functional coverage metrics can be specified in SystemVerilog Assertions though the definition of user-defined coverage points using the cover directive (see Section 4.5.1.4). In addition, simulation vendors typically provide the collection of statistics on the assertions, thus yielding feedback about the thoroughness of the simulation.²¹ Formal tools furnish coverage information such as unreachability and bounds of a proof search. Other tools provide statistics based on assertion monitoring of the whole design such as minimum sequential distance, and assertion density.

For example, assertions that monitor the logic controlling a FIFO queue could be used to capture the FIFO full condition, and verify that corner cases were exercised. Functional coverage is achieved by monitoring during verification whether the assertions were exercised. If the assertions weren’t exercised, that indicates that the logic (e.g., queue overflow) was not verified. Note that functional coverage is different from code coverage. Code coverage is a technique for measuring how much of the source code of a computer program or hardware description has been executed by a test run or simulation.²² However, functional coverage addresses the functionality of the design. In essence, functional coverage provides at a higher-level of abstraction about the completeness of the verification process.

1.1.3.6 Generates Counterexamples to Demonstrate Violation of Properties

Formal verification tools use SystemVerilog Assertions constraints (e.g., assume and restrict) and design properties to determine if the design meets those properties. If it does not, then those tools will generate counterexamples to demonstrate the conditions under which the rules were violated. This greatly facilitates the verification of designs (see Chapter 7).

Assumptions are used as restrictions by formal tools. Formal tools can give us the example for both cover and assertions to show how to achieve some coverpoint or how to get a violation. In such examples assumptions and restrictions from SVA need to be fulfilled. Assumptions are environment constraints, whereas restrictions are case specific. For example, a restrict statement would be used to confine an evaluation to one of several valid modes of operation. An assumption would be used to specify a required protocol response.

1.1.4 Can/Should Entire Functional Verification Task be Performed Using SVA?

A direct answer will be NO. However, as previously mentioned, SystemVerilog Assertions along with potential SystemVerilog constructs and supporting code will play a significant role in the definition of design requirements and in the verification process of designs. Use of SystemVerilog Assertions is strongly encouraged because Assertion-Based Verification was demonstrated to be very beneficial. SystemVerilog Assertions does not preclude the use of other proven techniques, such as accelerators, or the use of other languages, such as SystemC. However, it can supplement those techniques during the design definition and verification process. Assertions do not come for free though. It is important to understand where they provide the most value. In general, assertions are ideally suited for verification of control logic, but other techniques such as scoreboards may be more desirable for data.

²¹ EDA vendors may provide assertion coverage metrics including the number of times an assertion finished and failed, and how many times the assertion is triggered/checked.
1.1.5  Is SystemVerilog Assertions Solely Restricted to Applications that Use SystemVerilog?

Absolutely not! SystemVerilog Assertions can be used with designs written in any HDL, VHDL or Verilog. VHDL can be used in mixed mode simulation along with SystemVerilog Assertions. Section 4.7 expands these concepts by using real examples of binding SystemVerilog modules / checkers with assertions to VHDL and SystemVerilog designs. Binding to VHDL is non-standard, however many simulators have extended the standard to accommodate it.

Why would such an approach be beneficial? Because SystemVerilog Assertions is part of SystemVerilog, and a verification module that can make use of both the powerful constructs of the language and the flexibility and ease of use of the assertions.

1.2  SystemVerilog Assertions Goals

Ensuring that a design's implementation satisfies its specification is the foundation of hardware verification. Key to the design and verification process is the specification. Yet historically, the process of defining specifications consisted of creating a natural language description of a set of design requirements. This form of specification is both ambiguous, and in many cases, unverifiable due to the lack of a standard machine executable representation. Furthermore, ensuring that all functional aspects of the specification have been adequately verified (that is, covered) is problematic.

The SystemVerilog Assertions aspect of SystemVerilog was developed to address these shortcomings and help in the specification and verification processes. SystemVerilog Assertions shares many elements from other parts of the SystemVerilog language (such as expression syntax); however, it introduces a substantial number of new constructs intended to represent complex temporal behavior and verification intent that is unique to assertions. The assertions are integrated in SystemVerilog with very similar syntax. No longer does an engineer need to learn (1) a design language, (2) an HVL (hardware verification language), and (3) an assertion language. They are all part of the same language. SystemVerilog Assertions give the design architect a standard means of specifying design properties using a concise syntax with clearly defined formal semantics. Similarly, it enables the RTL designer to capture design intent in a verifiable form, while enabling the verification engineer to validate that the implementation satisfies its specification through dynamic (that is, simulation) and formal verification means. Furthermore, it provides a means to measure the quality of the verification process through the creation of functional coverage models built on formally specified properties. It also provides a standard means for hardware designers and verification engineers to rigorously document the design specifications with a machine-executable language. SystemVerilog Assertions were specifically developed to fulfill the following general hardware functional specification requirements:

- Easy to learn, write, and read.
- Concise syntax.
- Rigorously well-defined formal semantics.
- Expressive power, permitting the specification for a large class of real world design properties.
- Use of known and efficient underlying algorithms in simulation, as well as formal verification.
- Integrated into SystemVerilog environment, including scheduling of assertions, system tasks, functions, and data structures.