CHAPTER 1

INTRODUCTION

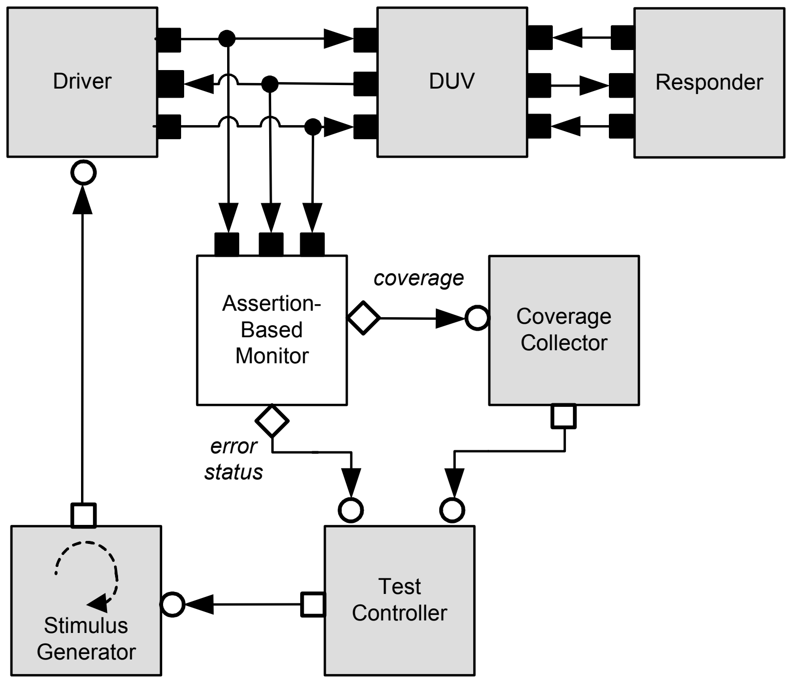
A quick skimming of this book might lead you to think that this is just another book about assertion-based verification. Or perhaps you might think this is a book about assertion patterns that will provide a knowledge base of property set examples for many common design components. But this book is really about process, specifically, a systematic process for creating reusable assertion-based verification components, which we refer to as *assertion-based IP.* Assertion-based IP is one form of verification components often lumped under the broader term *verification intellectual property* (VIP). The unique characteristic of assertion-based IP is that it is a reusable property set that takes advantage of assertion and coverage directives and easily integrates with other verification components within a verification environment. Reuse is achieved across multiple design implementations and multiple verification processes. The general relationship of assertion-based IP to VIP will be discussed in Chapter 2, "Definitions and Terminology."

1.1 Assertion-Based IP

assertion-based IP is more than a comprehensive set of related assertions

Figure 1-1 illustrates a typical assertion-based IP verification component within a contemporary transaction-level simulation environment. A key observation is that our assertion-based IP is much more than a comprehensive set of related assertions. It is a full-fledged reusable and configurable verification component, which is used to detect both interesting and incorrect behaviors. Upon detecting interesting or incorrect behavior, the assertion-based IP alerts other verification components (connected to its analysis ports), which are responsible for taking appropriate action.

Figure 1-1 A typical network of verification components



Guiding principles

The guiding principles we embrace when creating an assertion-based IP monitor are:

- *modularity*—assertion-based IP should have a clear separation between detection and action
- *clarity*—assertion-based IP should be written initially focusing on capturing intent (versus optimizations)

Modularity facilitates reusable verification components and simplifies maintenance. By clearly separating detection from action the assertion-based IP is not restricted in the way it must be used. This enables support for many different types of verification components and testbench architectural features (such as error injection within a simulation environment).

In section 1.2.3 “Simulation vs. formal verification,” we will talk more about our philosophy of focusing on capturing intent versus optimizations when creating assertion-based IP.

Why assertion-based IP? Certainly imperative forms of verification IP (that is, procedural code) delivered as an executable module is not a new concept. However, one problem with this form of verification IP is that it becomes necessary to identify the verification target point to achieve reuse across different tools (which is often not possible).

For instance, Example 1-1 demonstrates a portion of some procedural code used to check that two `grant` signals are never active at the same time.

Example 1-1 Imperative check for mutual exclusive grants

```
module my_verification_IP (. . . )

 always @(posedge clk) begin
    . . .
    if (grant[0] & grant[1]))
      $display (“Mutex violation for grant[0] and grant[1]”);
    . . .
  end

endmodule
```

This code could not be synthesized into a hardware monitor since there is not a clear verification target point (that is, not every `$display` would be a candidate target for hardware monitoring). Similarly, a formal tool would not know the verification target point for its proof.

Example 1-2 Declarative check for mutual exclusive grants

```
module my_verification_IP (. . . )
  . . .
  property p_mutex (g0, g1);
   @(posedge clk) !g0 & !g1;
  endproperty

  assert property (p_mutex(grant[0],grant[1]));
  . . .
endmodule
```

The previous example could be re-coded with a declarative `assert` directive, as illustrated in Example 1-2.

Hence, assertion-based IP has a clear advantage over traditional forms of verification IP monitors because they enable efficient reuse across a wide range of verification tools.

Assertions enable new verification technologies

Within the last decade, we have seen significant interest in assertion-based techniques, which have moved beyond the bounds of academic discussions into the realm of industry application. With the recent standardization of assertion and property languages, such as the IEEE Property Specification Language (PSL) [IEEE 1850-2005] and SystemVerilog Assertions (SVA) [IEEE 1800-2005], we have seen the development of new assertion-based design and verification technologies, which have opened new EDA markets by providing automated solutions to many verification challenges (for example, debugging, coverage specification, and intent validation).

Industry focus has predominately been on implementation-level assertions

With the growing interest in assertion-based techniques, a myriad of books have emerged that focus predominately on teaching syntax and semantics for these new assertion language standards. Often the examples provided in these books focus on implementation-level assertions. That is not to say it is worthless to add implementation-level assertions to a design. On the contrary, plentiful industry-published data and project statistics tout their benefits [Foster et al., 2004] .

Moving the assertion discussion up to a higher level

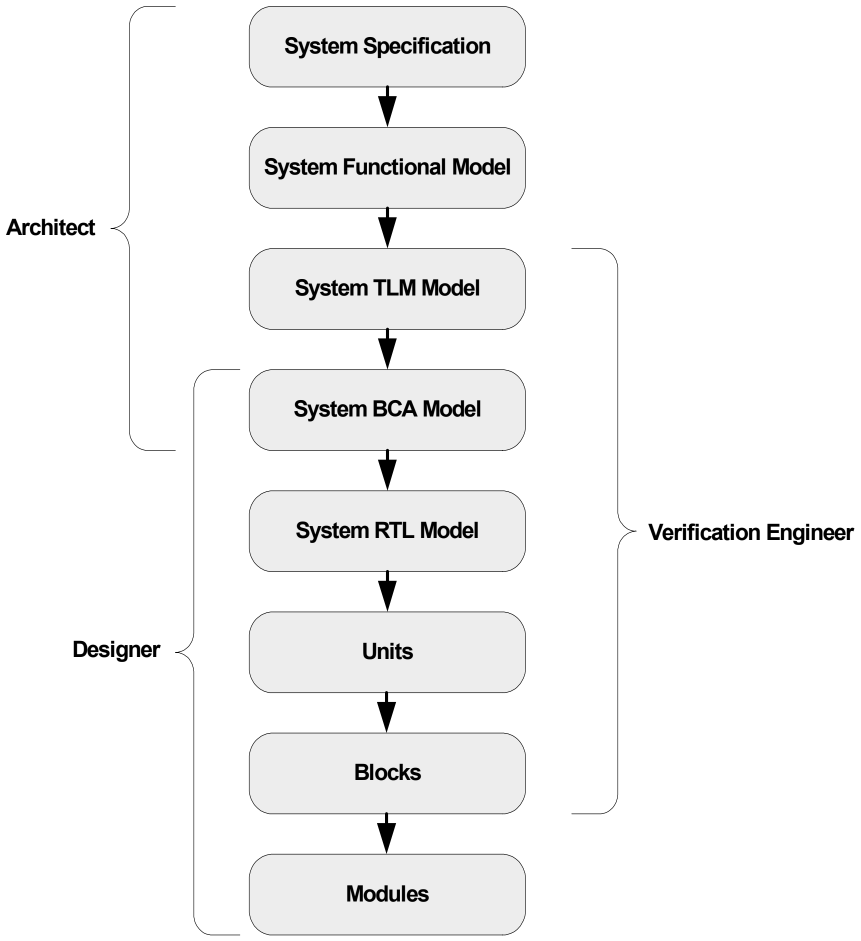
One of the criticisms we received after completing our *Assertion-Based Design* [Foster et al., 2004] book, was that our assertion patterns and cookbook examples tended to focus predominately on implementation-level assertion examples. Although this was helpful for the design engineer (it provided an aid to learning how to write embedded RTL assertions) it was of less value to the verification engineer who was interested in validating higher-level or black box behaviors. Hence, the focus of this book is to bring the assertion discussion up to a higher level and introduce a process for creating effective, reusable, assertion-based IP, which easily integrates with the user's existing verification environment (that is, testbench infrastructure).

1.1.1 Stakeholders

Our personal experience has demonstrated repeatedly that assertions added at any level of hierarchy (or abstractions) will clearly benefit verification by reducing debugging time while clarifying design intent. Certainly multiple stakeholder within the design and verification process can potentially contribute to the assertion development process—thus reducing ambiguities while improving observability.

Figure 1-2 illustrates a typical design refinement process through various levels of abstraction and the stakeholders associated with each level. Adoption of assertions in the industry, at the time of this writing, has predominately occurred within the block and module level, which we refer to as implementation assertions. This phenomenon is partially due to the lack of effective guidelines for assertion use at higher levels of design hierarchy (or abstraction) and confusion about which stakeholders should contribute to the assertion development process.

Figure 1-2 Stakeholders and levels of abstraction



Architects, verification engineers, and designers all contribute to the assertion development process

Although the architect could contribute to the assertion development effort by defining global properties (derived from the architecture and micro-architectural specification) that must hold across multiple possible implementations, the design engineer contributes by writing internal white-box assertions derived from the implementation. In addition, the verification engineer contributes by developing assertions specifying correct interface behavior between units and between blocks. The verification engineer also contributes by developing black-box, end-to-end assertions across design components.

Understand what you intend to specify before you write your assertions

Although it is easy to identify the various stakeholders and roles they play in the assertion development process, these stakeholders often lack a process for systematically developing their assertions. In fact, a classic mistake many engineers make when first adopting assertion-based techniques is to jump into coding assertions too soon—without fully understanding the behavior they are trying to specify. This ad hoc approach leads to incomplete (or incorrect) property sets and non-reusable assertion-based IP. We address this problem by introducing a systematic process in which we create a natural language list of properties prior to coding the assertions, which is a fundamental step in our process for creating assertion-based IP.

1.1.2 Levels of abstraction

When applying assertion-based techniques, it is important to first understand where they can be effectively applied. For example, Figure 1-2 illustrates a typical design refinement process.

System specification refined to a system functional model

The flow begins with a *system specification*, which is typically a natural language properties document. There has been recent interest in executable forms of system specification, such as UML [Martin 2002]. The first refinement of the system specification is often a *system function model* to explore the proposed algorithm, which is often written in C or C++. At this point, hardware-software partitioning and architectural mapping decisions have not been made.

System transaction-level model

The *system transaction-level model* (TLM) is generally an untimed (or partially timed) model, and it is created after architectural mapping. This model is often used for firmware development, system and architectural performance analysis, and software development. The TLM is often further refined into a *bus cycle-accurate* (BCA) model as architectural decisions begin to gel.

RTL model

RTL refinement occurs next, and the system is partitioned into multiple *units*. Each unit is partitioned into *blocks*. Each block is partitioned into *modules* consisting of RTL code.

Natural language list of properties

Certainly a natural language list of properties can (and should) be developed at all levels of abstraction. However, today's assertion language standards lack the proper formalism necessary to express properties at all levels of abstraction illustrated in Figure 1-2. For example, within an *untimed* transaction-level model, one concurrent transaction might overlap with a different transaction. That is, it could begin and end in the middle of a different untimed transaction. Existing assertion language standards lack the semantics (and syntax) necessary to express assertions related to this type of transaction behavior. Researchers have proposed solutions to the TLM assertion specification problem, which might result in semantic and syntactical extensions to existing standards [Ecker et al., 2006]. However, successful assertion specification today in an industry setting occurs at and below the BCA abstraction level, illustrated in Figure 1-2. We will demonstrate how to develop assertion-based IP that can be reused within a transaction-level testbench using existing standards by introducing what is essentially an abstraction converter between a timed RTL model and untimed transaction environment.

1.1.3 Reuse

Design and verification productivity drive reuse adoption

Many forces at play contribute to a gap between what we can manufacture (silicon capacity) and what we have time to design. Furthermore, these forces contribute to the gap between what we can design and what we have time to verify. Intellectual property (IP) offers the promise of filling these gaps by increasing both design and verification productivity. In fact, recent studies, such as the International Technology Roadmap for Semiconductors (ITRS) [ITRS 2003], indicate that 75 percent of design productivity improvement will come from IP reuse and 25 percent from improved EDA tools, flow, or methodologies. A key factor in both design and verification economics is IP reuse. Yet, without systematic processes for development, IP is often delivered with poor quality, and it is difficult to use or incompatible with other design and verification components. While industry guidelines for design reuse have been published [Keating and Bricaud 2002] few guidelines for verification IP reuse exist [Glasser et al., 2007] [Wilcox 2004] [Bergeron et al., 2006], and certainly (to the best of our knowledge) no guidelines exist specifically focused at assertion-based IP. Hence, we believe that a systematic process and set of guidelines for creating effective reusable assertion-based IP are sorely needed. And that is the focus of this book.

1.2 Properties and assertions

The recent flurry of interest in assertion-based techniques has prompted considerable published work on the subject. Often, the authors of these publications interchange the terms *property* and *assertion*, which leads to confusion. For our discussion, we informally define a property as follows:

Definition•1.1 ***Property***—a statement of expected behavior.

For example, the statement, *grant[0] and grant[1] are mutually exclusive* is an example of a property, which is actually a partial specification for an arbiter. Notice that we have not stated how we intend to use this property during verification. For example, we might choose to use this property as a constraint specification, which is a requirement on the environment, and *assume* the property during verification. In this case, we want to eliminate traces that violate our constraint. Or, we might choose to use this property as a coverage specification, and *cover* the property such that the verification tool notifies us at the particular point in which the coverage specification holds on a trace. Or, we might choose to use the property as a design requirement specification and *assert* that the property holds on all traces of a design during verification.

We informally define an assertion as follows:

Definition•1.2 ***Assertion***—an implementation of a property that is evaluated or executed by a tool to validate design intent.

We use an assertion as a target during verification (that is, a checker for simulation or a proof obligation for formal) to help us identify and isolate unexpected behavior.

1.2.1 Languages

To evaluate our natural language property *grant[0] and grant[1] are mutually exclusive* during verification, we must express the property in a machine executable form (that is, an assertion language). Two alternative languages we can chose from to express this property are: *SVA*, which is an assertion sublanguage within IEEE 1800-2005 Std SystemVerilog or *PSL*, which is the IEEE 1850-2005 Std Property Specification Language.

SVA and PSL In this book, we have chosen SVA to demonstrate our examples. We could express our examples just as easily in PSL—allowing reuse of our property set between simulation and formal verification, and you might chose to create your property set using PSL [Eisner and Fisman 2006]. However, we also want to demonstrate other aspects of creating reusable verification IP (such as proper handling of error injection), which requires explicit communication between an assertion (or cover directive) and other verification components within a simulation environment. This communication is easily accomplished by using assertion and coverage action blocks along with analysis ports defined within our assertion-based IP interface. The following chapters discuss details about connecting assertion-based IP with multiple verification components.

1.2.2 Libraries

While selecting a property and assertion language standard is an integral step for adopting assertion-based techniques, it is not the entire solution. Methodology is equally important to effectively applying ABV. In fact, without enforcing a consistent methodology across multiple design and verification engineers (for example, a consistent means of reporting errors, enabling or disabling assertions, and assigning actions performed upon error detection), the *ad hoc* use of assertions can end up being unmanageable and disruptive to the overall verification process.

Assertion libraries provide a means for ensuring that a consistent ABV methodology is employed across an entire project's verification flow. Furthermore, libraries provide an ideal means to encapsulate configurable property sets that can be preverified, which makes them ideal for assertion-based IP reuse.

The Open Verification Library

As we discussed in the previous section, assertion specification can occur at multiple levels of hierarchy and multiple levels of abstraction. The same holds true for libraries used to create assertion-based IP. For example, the Accellera OVL [Accellera OVL 2007] incorporates a consistent and systematic means of specifying RT-level implementation properties structurally through a common set of library elements (that is, assertion monitors). The OVL library elements act like a template that lets designers express a broad class of assertions in a common, familiar RTL format. Furthermore, the OVL capitalizes on the various emerging property and assertion language standards by delivering multiple flavors of the library, implemented either in pure Verilog '95, PSL, or SVA. This versatility is important to design IP providers who must deliver design IP code to a diverse group of customers, each with their own restrictions in terms of language support [Foster et al., 2006a]. The OVL enables the IP provider to instantiate an assertion checker as a module within the design IP, and then allows the users to link in an appropriate library that will work best with the verification infrastructure and environment. For example, returning to our natural language property *grant[0] and grant[1] are mutually exclusive* example, the IP provider would instantiate an OVL checker into its RTL as demonstrated in Example 1-3.

Example 1-3 Instantiated OVL checker

```
assert_always mutex_grants (clk, reset_n, !(grant[0] & grant[1]));
```

Example 1-4 illustrates the various supported implementations of the OVL `assert_always` monitor. The `test_expr` (in Example 1-4) is a formal parameter that receives the actual parameter `!(grant[0] & grant[1])` (from Example 1-3) when `assert_always` is called.

Example 1-4 assert_always OVL implementation example

```
`include "std_ovl_defines.h"

`module assert_always (clk, reset_n, test_expr);
  parameter severity_level = `OVL_ERROR;
  parameter property_type = `OVL_ASSERT;
  parameter msg="VIOLATION";
  parameter coverage_level = `OVL_COVER_ALL;
  input clk, reset_n, test_expr;

`ifdef OVL_VERILOG
  `include "./vlog95/assert_always_logic.v"
`endif // OVL_VERILOG

`ifdef OVL_SVA
  `include "./sva31a/assert_always_logic.sv"
`endif // OVL_SVA

`ifdef OVL_PSL
  `include "./psl11/assert_always_psl_logic.v"
`endif // OVL_PSL

`ifdef OVL_PSL
//Do not add "`endmodule" as this will be added in the included file
`else
  `endmodule
`endif
```

Without having to create and supply multiple versions of the IP supporting all the different assertion languages, the IP provider ships its IP with OVL monitors instantiated, and the customer selects an appropriate version of the OVL to link into its verification environment. For example, the SVA flavor of the OVL `assert_always` can contain the code

Example 1-5 SVA flavor of OVL (assert_always_logic.sv)

```
`ifdef ASSERT_ON
assert_always:
  assert property(
    @( posedge clk) disable iff ( !reset_n)
      (test_expr)
        else sva_checker_error("");
`endif
```

Higher level libraries

In this book, we demonstrate a process for creating complex library verification components we call assertion-based IP. Unlike the OVL, our examples target behaviors at higher levels of the design hierarchy and a higher level of abstraction (for example, a comprehensive set of assertions that specify the proper behavior of an SDRAM memory controller). Certainly, these more complex assertion-based IP verification components can be constructed by encapsulating a set of OVL with additional modeling code inside a single monitor. However, we have chosen to take advantage of SVA's action blocks to demonstrate the assertion-based IP examples in this book. Action blocks allow us to control the separation of detection from action by taking advantage of analysis ports (discussed in Chapter 3, "The Process") within the assertion-based IP architecture.

1.2.3 Simulation vs. formal verification

Our assertion-based IP development philosophy

With the advent of the OVL standard, the industry saw its first alternative solution for "specifying once" and then leveraging assertion specification over multiple verification processes—such as traditional simulation and formal verification tools [Foster and Coelho 2001]. In fact, this is one of the value propositions you might have heard from various EDA vendors promoting assertion-based techniques. While this is certainly true for many simpler white-box implementation assertions, it has been our experience that higher-level, black-box forms of assertions describing behavior across a block, often require abstractions or advanced coding techniques to achieve a complete formal proof. These advanced techniques are beyond the scope of this book. Our philosophy toward creating assertion-based IP is that you should first focus on capturing the intent of the design in a clear set of assertions suitable for simulation, and then optimize any of the problematic assertions you encounter only if you experience problems while attempting proof—and only if there is a clear return on effort to achieve a complete proof. You can certainly reuse assertions written for simulation as bug-hunting targets with formal verification without optimization.

1.3 Who should read this book?

We have created this book for many different audiences. For example:

- The novice in assertion-based techniques, who might be an electronic engineering student learning about design and verification and interested in learning from practical examples
- The academic instructor, who is looking for real assertion examples beyond the typical toy circuit examples
- The experienced design or verification engineer who is considering adapting ABV on a future project and wants to evaluate processes
- The design or verification manager whose goal is to improve the capability maturity of their organization through state-of-the-art industry best practices and processes
- The system architect who wishes to improve on communicating design intent to other stakeholders within a project team

- The verification IP provider who is looking for a systematic process to develop assertion-based IP

1.4 Book organization

Creating Assertion-Based IP is organized into chapters that you can read sequentially for a full understanding of our topic. Or you might wish to focus on a particular area of interest and visit the remaining chapters to supplement your understanding. We have allowed repetitions to support the readers who prefer to browse for information as well as readers who intend to read cover-to-cover and then revisit sections of particular personal relevance. Highlights of the contents follow.

Chapter 2, "Definitions and Terminology"

Chapter 2, "Definitions and Terminology" provides definitions of terms and acronyms used throughout the book and a verification IP framework in terms of verification components found in a contemporary testbench.

Chapter 3, "The Process"

Chapter 3, "The Process" introduces a systematic set of steps recommended for creating assertion-based IP. In addition, this chapter discusses project-specific concerns for effective assertion-based IP integration.

Chapter 4, "Bus-Based Design Example"

Chapter 4, "Bus-Based Design Example" sets a framework for our discussion in the remaining chapters of the book by introducing a typical SoC bus-based design example, which consists of various common functional design components. We then demonstrate our assertion-based IP creation process on many of these design components in the remaining chapters of the book.

Chapter 5, "Interfaces"

Chapter 5, "Interfaces" demonstrates our assertion-based IP creation process for various bus interfaces. On-chip bus protocols form the foundation for many of today's design reuse strategies. This chapter discusses assertion-based IP for serial, nonpipelined, and pipelined-bus interfaces.

Chapter 6, "Arbiters"

Chapter 6, "Arbiters" demonstrates our assertion-based IP creation process for various arbiters. Arbiters are critical components in systems containing shared resources. For example, for a bus-based design where more than one bus master might drive a shared bus, arbitration prevents dropping or corrupting data transported through the bus.

Chapter 7, "Controllers"

Chapter 7, "Controllers" demonstrates our assertion-based IP creation process on typical controllers. While busses serve as the framework for platform-based SoC designs, it is controllers that provide the operational brains. Controllers span the spectrum of design, from lower-level control units embedded in complex design components (such as a simple one-hot encoded control unit), to very complex controllers (such as a graphics controller, memory controller, and an embedded CPU's cache control unit).

Chapter 8, "Datapath"

Chapter 8, "Datapath" demonstrates our assertion-based IP creation process on blocks involving datapaths. In this book, we have not limited our discussion to written assertions that are only amenable to formal verification. Hence, we demonstrate our assertion-based IP creation process on both data transportation and data transform blocks (many of which are not good candidates for formal verification using model-checking).

Appendix A, "Quick Tutorial For SVA"

Appendix A, "Quick Tutorial For SVA" provides a basic introduction to SystemVerilog Assertions. Specifically, we discuss the constructs we use in this book.

Appendix B, "Complete OVM/AVM Testbench Example"

Appendix B, "Complete OVM/AVM Testbench Example" provides a complete transaction-level testebench with an assertion-based IP example. In addition, it provides a basic introduction to the Open Verification Methodology (OVM) base-class library components we reference in this book.

1.5 Summary

This chapter presented the motivation for creating assertion-based IP and the stakeholders involved in the process. A unique characteristic of assertion-based IP is that it is a reusable property set, which takes advantage of assertion and coverage directives and easily integrates with other verification components in a verification environment. Reuse is achieved across multiple design implementations and multiple verification processes (for example, simulation and formal verification).