CHAPTER

2

DEFINITIONS AND TERMINOLOGY

We exist in a multilanguage world where our success often depends on our ability to communicate intent and eliminate ambiguities. Yet even within a common discipline, such as engineering, the problem and solution space is often quite varied (due to historical, cultural, and technical reasons), which leads to a difference in terminology that results in misunderstandings. To ensure you (the reader) are on the same page with us (so to speak), we have included this chapter, which provides definitions for the terminology we use throughout the remainder of the book.

Chapter organization

This chapter is divided into two main sections. The first section builds a framework for our discussion by introducing common verification components found within contemporary simulation environments. Understanding how the various verification components potentially interact and the communication channels required to connect these components is critical as we architect our assertion-based IP solution.

The second section of this chapter provides a set of definitions for many terms used throughout this book. In addition, we spell out a list of common acronyms related to our topic. In this section, we discuss architectural aspects of a contemporary transaction-level testbench. We chose the

Advanced Verification Methodology [Glasser et al., 2007], which is a subset of the newly formed OVM, as the basis for our discussion because the source code for the OVM library is openly available and can be freely downloaded at *http://www.mentor.com/go/cookbook*. Assuredly, there are other testbench base-class libraries available, and we encourage you to choose one that you feel comfortable with when creating your assertion-based IP. The general ideas, processes, and techniques we present in this book for creating assertion-based IP are easily extended to the specific implementation details of other verification environment methodologies, such as the VMM [Bergeron et al., 2006] and the eRM [2005].

2.1 Notation

Connecting separate verification components through well defined interface is a key tenet of our assertion-based IP methodology, and is reflected in the example notation we use throughout the book. Hence, in this section we define the graphical notation, which includes: *components, interfaces, interconnects, channels, and analysis ports*.[1]

2.1.1 Components

A component is an instantiable object in SystemVerilog, such as a module, interface, program block, or class. For our discussion, we represent a component as a box, as illustrated in Figure 2-1.

Components often have free running threads. Sometimes, the location of threads in a design or testbench is important to understanding the architecture of the design. To show a

1. Notation discussion based on Advanced Verification Methodology (AVM) concepts. See [Glasser et al., 2007].

component that has one or more threads we use a circular arrow, as illustrated in Figure 2-2.

Figure 2-1 Component symbol

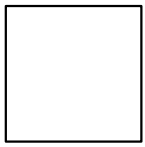
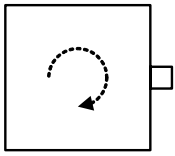


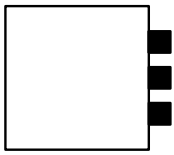
Figure 2-2 A component with a thread



2.1.2 Interfaces

Interfaces are the externally visible connections to components. All of a component's behavior is accessible and visible only through its interfaces. First, is the familiar pin interface, as illustrated in Figure 2-3.

Figure 2-3 A component with a pin interface

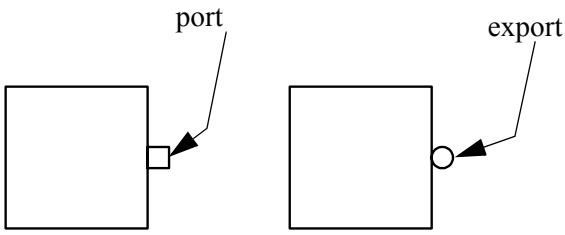


The black boxes on the right side of the component represent pins.

Whereas pin interfaces move data represented at the timed bit level between components, transaction interfaces move high level untimed (or partially timed) data between components.

Figure 2-4 represents two variations of transaction interfaces: a *port* and an *export*. The component on the left has a transaction port and the component on the right has an export. An export makes an interface visible, and a port is a requirement to connect to an export. A good way to think about transaction ports is as a set of unresolved function calls that are resolved by exports. Ports and exports are complements of each other; ports connect to exports. You cannot connect an export to an export, nor a port to a port.

Figure 2-4 A component with a transaction-level interface

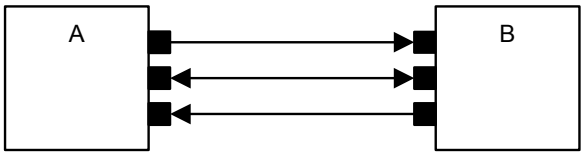


The port/export notation identifies the flow of control between components. Since a port interface calls functions on an export, flow of control moves from ports to exports.

2.1.3 Interconnect

Just like with traditional schematics, we use lines between interfaces to show the interconnection amongst components. The addition of arrow heads allows us to represent data flow.

Figure 2-5 Pin-level data flow



Arrows between pins show the direction that data flows between components. The figure above shows, from top to

bottom, flow from A to B, bi-directional between A and B, and flow from B to A.

Figure 2-6 Transaction data flow

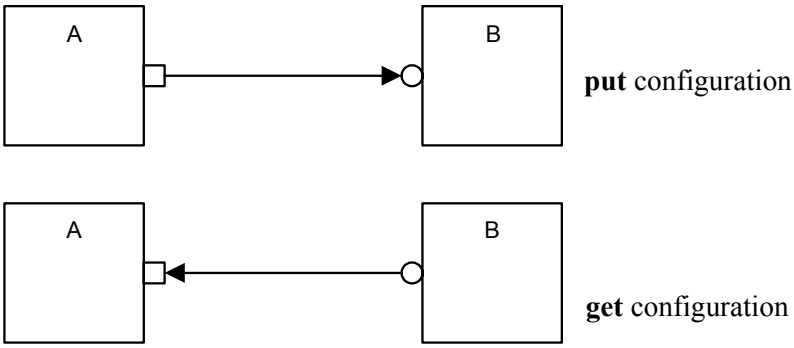


Figure 2-6 illustrates two configurations, each with the same transaction interfaces, but with different data flow. In both configurations, a function in B is invoked by A. A initiates activity in B. A is the *initiator* and B is the *target*. In the top configuration, A moves data to B. This is called a *put* operation. In the bottom configuration, A moves data from B back to itself. This is called a *get* operation.

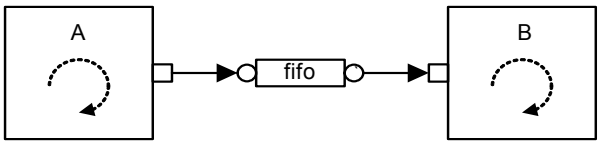
2.1.4 Channels

Transaction level components often communicate through channels. A channel is a component that defines the semantics of the communication. One of the most common channel usage is a FIFO. FIFOs are used to throttle communication between two transaction level components. To show this in a netlist, we show a small box between components to represent the FIFO.

A FIFO, as with other communication channels, exports an interface. However, in the interest of keeping the diagram uncluttered, the circles on the channel exports are optional and often omitted.

Figure 2-7 shows two components, each with their own thread, and each with a transaction port that connects to an intervening channel. Component A puts transactions into the FIFO channel and component B gets transactions from the same channel. The data flow arrows in addition to the transaction ports tell us which components are doing gets and which are doing puts. A has a thread, a transaction port (as opposed to an export), and an arrow leading away from it. That tells us that A is putting transactions into the channel. B also has a thread and a transaction port, but the data flow arrow is leading into the component instead of away from it. That tells us that B is getting transactions from the channel.

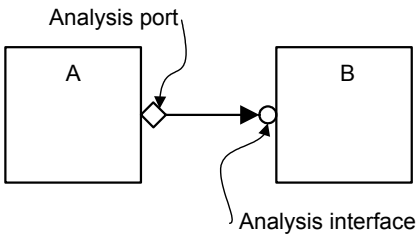
Figure 2-7 Two components communicating through a FIFO



2.1.5 Analysis ports

Analysis ports (illustrated in Figure 2-8) are a kind of transaction-level port used for communicating analysis information (for example, coverage data or assertion error status) between components. The symbol for an analysis port is a diamond. Analysis ports are connected to a component with an analysis interface. This could be an analysis FIFO or a component with an analysis interface.

Figure 2-8 Analysis port connected to an analysis interface



2.2 Verification component description

Table 2-1 provides a list and description for many common verification components found in a contemporary verification environments.

Table 2-1 Contemporary testbench verification components

Name	Description
Control	
`simulation controller`	The simulation controller is the decision maker within the testbench. It completes the loop from stimulus generator through driver, monitor, scoreboard, and coverage collector.
Analysis	
`coverage collector`	A coverage collector is responsible for recording observed behaviors defined in an associated coverage model.
`scoreboard`	A scoreboard is one type of verification component used to determine the end-to-end correctness of the DUV. Input stimulus entering the DUV is stored inside the scoreboard, which will then be used/transformed for comparison against the DUV's output response.
Environment	
`stimulus generator`	A stimulus generator is a verification component that generates stimulus. For example, in a contemporary testbench, a stimulus generate might generate either directed or random transaction-level stimulus. Hence, it would contain the algorithms and constraints used to generate stimulus.
`scenario generator`	A scenario generator is a form of stimulus generator that generates a stream of a directed sequence of transactions that is intended to perform a specific well-defined function on the DUV.
`master`	A master is a bi-directional verification component that sends requests and receives responses. A master initiates activity and may use a response to determine the next course of action.

Name	Description
`slave`	A slave is a bi-directional verification component. Slaves reply to requests and receive responses, they do not initiate activity.
Transactors	
`driver`	A driver verification component converts the untimed or partially timed transaction-level stimulus into timed pin-level activations on the DUV.
`responder`	A responder verification component is similar to a driver. It connects to a bus and will drive timed pin-level activity on the bus. The responder responds to activity on the bus rather than initiating it.
`monitor`	A monitor is a passive verification component. It observes the pins of the DUV and converts the timed pin-level activity into a stream of transactions for use by other components within a testbench.

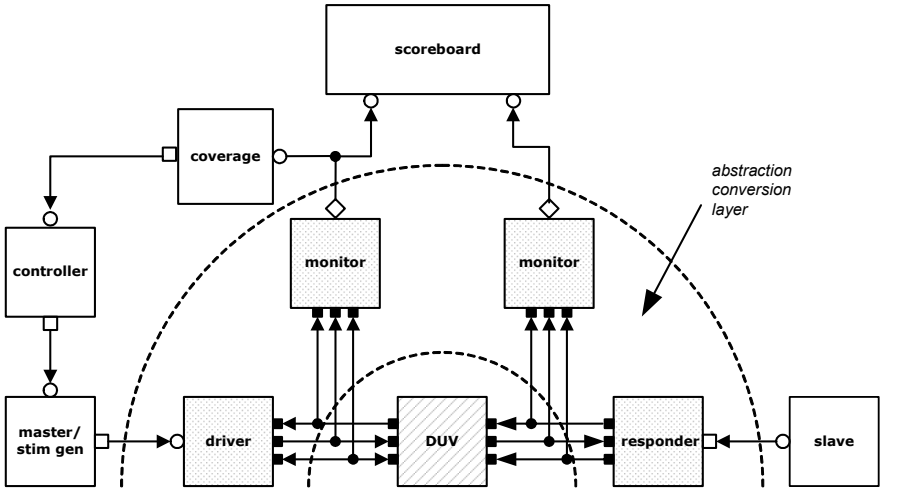
2.3 Verification component organization

Today's hardware designs are modular in their composition and can be viewed as a network of various design components. Similarly, contemporary testbenches are also modular in their composition and can be viewed as a network of various verification components. In fact, it is this property of modularity, in terms of well-defined behavior and interfaces for these various components, that simplifies the construction, debugging, and maintenance of today's complex systems.

Network of verification components

Concentric testbench organization. Figure 2-9 illustrates an example of various common verification components found in a typical testbench. In our example, you will note that our testbench architecture is in multiple layers. The bottom layer is the register transfer level design under verification (DUV). All pin-level activity at this layer is timed (that is, either coarse-grain cycle accurate timing or fine-grain signal accurate timing).

Figure 2-9 A typical network of verification components



Above the RTL layer (labeled abstraction conversion layer in Figure 2-9) is a set of verification components known as *transactors*, which are used to convert untimed (or partially timed) streams of transactions into pin-level timed activity—and vice versa. All verification components above the abstraction conversion layer are at the transaction layer.

Hierarchy of verification components

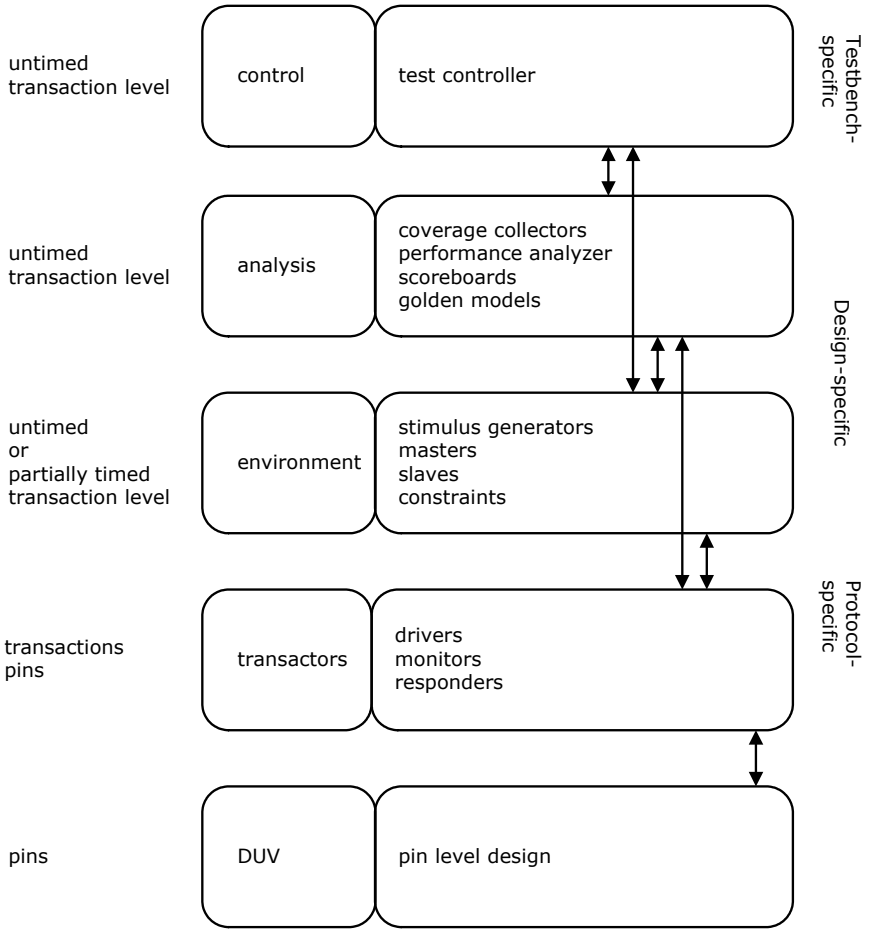
Hierarchical testbench organization. As an alternative, and for the purpose of our discussion, we view the architecture layers of a testbench as a hierarchy of verification components organized into sets of similar functionality as illustrated in Figure 2-10.

Modularity benefits

By organizing the testbench verification components into sets of well-defined functionality, and establishing clear channels of communication and interfaces between the layers we are able to achieve our goal of modularity, which has the following benefits:

- Enables reuse of verification components
- Facilitates quick enhancements and maintenance through localization
- Simplifies debugging

Figure 2-10 Architectural layers and component types



Control layer

The control layer contains a simulation controller verification component, which provides the main thread of a test and orchestrates the activity. Typically a simulation controller receives information from analysis components, which it then uses to determine when to complete or terminate a test and send information to environmental components. Transactors, such as monitors, can pass status to the simulation controller upon detecting errors, and the simulation controller will then take appropriate actions.

Analysis layer

The analysis layer contains a set of components that receive information about what is going on in the testbench and uses

that information to determine the correctness or completeness of the test. Common analysis layer verification components include scoreboards and coverage collectors.

Environment layer

The environment layer contains a set of verification components necessary to operate the DUV, such as stimulus and scenario generators as well as constraints.

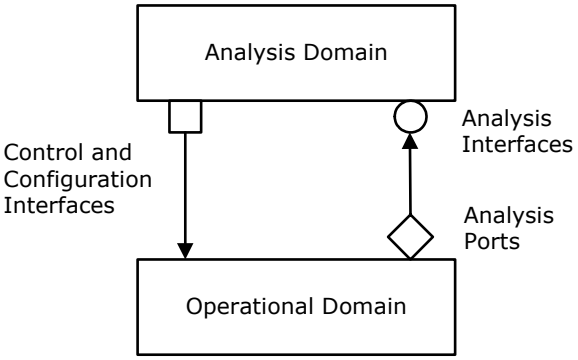
Transactor layer

The transactors layer contains a set of verification components that convert a stream of untimed or partially timed transactions into timed pin-level activity or vice versa. Transactors are typically characterized by having at least one pin-level interface and at least one transaction-level interface. Assertion-based IP (monitors) fall into this testbench architectural layer.

There are obviously other ways to view and organize the architectural layers within a testbench, for example [Bergeron et al., 2006] [eRM 2005], and you might have your own ideas for testbench architectural organization. Certainly we could spend hours debating the merits of various testbench organizations and architectural views. That is not the goal of this chapter. Our goal is to present an organizational view to set a framework for discussion throughout the remainder of the book. The ideas we present on how to effectively integrate assertion-based IP with other verification components are applicable (or easily extended) to alternative testbench architectural views.

Operational vs. analysis organization. To conclude our discussion on ways to view and organize verification components within a testbench, we partition the set of verification components into two domains, *operational* and *analysis,* as illustrated in Figure 2-11. The operational domain contains the set of verification components that operate on the DUV. The analysis domain contains the set of components that watch and analyze the operation.

Figure 2-11 Operational and analysis testbench domains



Analysis ports

Data moves from the operational domain to the analysis domain in a way that does not interfere with the operation of the DUV while it preserves event timing. You will see in our examples in the upcoming chapters that we accomplish this by means of a special communication channel called an *analysis port*. An analysis port is a unique transaction port in which a publisher (for example, an assertion-based IP module) broadcasts data to one or more subscribers (for example, both a coverage collector and a scoreboard). What is unique about this approach is that the publisher (and subscriber) need neither knowledge about the testbench architecture nor details about any of the verification components that are subscribing to its analysis port data. This preserves the localization property of modular design and makes the verification component truly reusable and interoperable. Essentially, the connection between the verification components is achieved by having each subscriber (that is, other verification components within the testbench) register with a particular publisher's (that is, in our case an assertion-based IP monitor) analysis port, which creates a list of subscriber interfaces for broadcasting data.

Chapter 3, "The Process" demonstrates how to create analysis ports as a SystemVerilog class, which is implemented as part of the Open Verification Methodology (OVM) base-class library. Additional details about the OVM base-classes we use in our examples are covered in Appendix B, "Complete OVM/AVM Testbench Example."

2.4 Definitions

In this section, we define the terminology common to the discipline of assertion-based IP. This section should be referenced whenever an unfamiliar word or phrase is encountered in this text. For a comprehensive list of terms, we recommend *Taxonomy for the Development and Verification of Electronic Systems* [Bailey et al., 2005]. In addition, *Functional Verification Coverage Measurement and Analysis* provides an excellent list of terms that are common to the language of coverage [Piziali 2004]. Finally, *ESL Design and Verification* provides an excellent taxonomy and definition of terms [Bailey et al., 2007]. Whenever possible, we have made an effort to align our definition of terms with these sources.

Table 2-2 Definition of terms

Terminology	Definition
`abstraction`	Describing an object using a model where some of the low-level details are ignored.
`assertion`	The implementation of a property evaluated or executed by a tool (see also property).
`assumption`	An environmental constraint.
`arbiter`	A design or verification component that manages shared resources.
`assertion coverage`	Observing an assertion evaluated or executed, passing or failing, and recording the possible paths of evaluation through the assertion.
`assertion-based IP`	A passive verification component, consisting of declarative properties and potentially additional procedural code, that monitors either bus interface activity or a design component end-to-end behavior.
`behavior`	A model of a system at any level of abstraction that potentially includes timing information.

Terminology	Definition
black box	A term used to define the amount of visibility internals of a block have and thus the amount of control an engineer has over that block for the purposes of verification. In the case of black box, no internals are visible. The opposite of black box is white box.
bus functional model (BFM)	A verification component that specifically focused on generating stimulus and checking results for correct bus signal activity. A BFM imitates correct bus signal characteristics.
constraint	Rules definition relationships between signals within a design; they can be combinatorial or sequential/temporal and can be used in pseudo-random generation and formal methods.
controller	A state-machine used to generate control signals that either activate or terminate various components within a design.
corner case	One or more data values or sequential events that, in combination, lead to a substantial change in design behavior. A corner case is often an exceptional (rare) condition that is difficult to predict.
coverage	A measure of how thoroughly a design has been exercised during verification to minimize the probability that latent errors remain.
coverage model	An abstract representation of device behavior composed of attributes and their relationships. Relationships may be either data or temporal in nature.
datapath	A datapath is hardware that either performs data processing operations (that is, transforms data) or buffers the flow of data (that is, transports data). It is one of two types of modules used to represent a digital system, the other being a control unit.
design component	A reusable implementation (that is, a model) of a well defined set of high-level or low-level functionality.

Terminology	Definition
dynamic verification	Demonstrating that a design conforms to its functional specification through execution. This form of verification relies on the progression of time to allow the design's internal logic to propagate through the design-specific values placed on the design's inputs at a given time. This algorithm requires some specific mechanism (for example, simulation, emulation, or prototype) to run the design and a specific methodology to apply stimulus and check the output of the design (verification environment) to verify correctness of the design.
intellectual property (IP)	A block of code that describes an aspect of a system, including its hardware, software, or the verification environment, which is reused between multiple designs or parts of a design. A complete IP for reuse should include its specification, verification plan, verification environment (dynamic and static components) and RTL implementation.
model	A way of capturing certain behavioral aspects of a system. A model normally involves the application of some amount of abstraction such that adequate performance can be obtained at a desired level of accuracy.
platform-based design	A reuse-intensive design style for embedded systems where large portions of the design are based on predesigned and preverified SoC design components. This is an integration oriented design approach that emphasizes systematic reuse for developing complex products based upon platforms and compatible hardware and software virtual components that are intended to reduce development risks, costs, and time to market.

Terminology	Definition
property	A statement of an expected behavior. For example, a liveness property says that something should eventually happen and is often called an eventuality. A safety property says that something should never happen and is often called an invariant. Liveness and safety properties define valid or invalid paths through the state space of a design.
protocol	A set of rules governing communication between design components.
protocol checker	A verification component (either procedural-based or assertion-based) that checks a set of rules of an interface and determines if violations of defined, acceptable behavior have occurred.
requirement	A requirement is: • A condition or capability needed by a user to solve a problem or achieve an objective. • A condition or capability that must be met or possessed by a system or a system component to satisfy a contract, standard, specification, or other formally imposed document. • A documented representation of a condition or capability as defined above.
static verification	The process of demonstrating that a design conforms to its functional specification through comparative analysis and proof, as opposed to design execution.
transaction	A transaction is an agreement, communication, or movement carried out between separate entities or objects. For example, a single transfer of control or data between two entities. Within a transaction-level testbench, a transaction is initiated via a function call (for example, `my_object.read()`).
verification component	An individual unit (that is, module or object) that are used as building blocks in the construction of a verification environment (such as a simulation testbench).

Terminology	Definition
verification IP (VIP)	An umbrella term for a reusable verification unit that is also the unique property of one party but may be licensed to another party (or can also be owned and used by a single party alone). Examples include BFM, assertion-based IP, protocol checker.
white box	A term used to define the amount of visibility and or control an engineer has into a block for the purposes of verification. In this case all internals are visible. The opposite of this is black box.

2.5 Acronyms

Table 2-3 provides a list of common acronyms related to the field of assertion-based IP, many of which are used throughout the book.

Table 2-3 Acronyms

Acronym	Meaning
AHB	AMBA Advanced High-performance Bus
AMBA™	Advanced microcontroller bus architecture
APB	AMBA Advanced Peripheral Bus
AVM	Advanced Verification Methodology
BFM	Bus functional model
DUV	Design under verification
EDA	Electronic design automation
eRM	***e*** reuse methodology
FIFO	First in first out
FSM	Finite state machine
HDL	Hardware description language

Acronym	
HVL	High-level verification language
I^2C	Inter-Integrated Circuit
IP	Intellectual property
OVL	Open Verification Library
OVM	Open Verification Methodology
PSL	Property Specification Language
RTL	register transfer-level
SDRAM	Synchronous dynamic random access memory
SoC	System on chip
SVA	SystemVerilog Assertions
TLM	Transaction-level model
VIP	Verification IP
VMM	Verification Methodology Manual

2.6 Summary

We created this chapter to establish a common language between us (the authors) and you (the reader). The first section built a framework for our discussion by introducing common verification components found within contemporary simulation environments (such as the OVM). We believe that a solid understanding of how various verification components potentially interact and the communication channels required to connect these components is critical to properly architect an assertion-based IP solution.

The second part of this chapter provided a set of definitions for many terms used throughout this book. In addition, we spelled out a list of common acronyms related to our topic.