
Applied Assertion-Based Verification: An Industry Perspective

Applied Assertion-Based Verification: An Industry Perspective

Harry Foster

Mentor Graphics Corporation

Plano, Texas

USA

Harry_Foster@mentor.com

now

the essence of **know**ledge

Boston – Delft

Foundations and Trends[®] in Electronic Design Automation

Published, sold and distributed by:

now Publishers Inc.
PO Box 1024
Hanover, MA 02339
USA
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:

now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is H. Foster, Applied Assertion-Based Verification: An Industry Perspective, Foundations and Trends[®] in Electronic Design Automation, vol 3, no 1, pp 1–95, 2008

ISBN: 978-1-60198-218-6

© 2009 H. Foster

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, mechanical, photocopying, recording or otherwise, without prior written permission of the publishers.

Photocopying. In the USA: This journal is registered at the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923. Authorization to photocopy items for internal or personal use, or the internal or personal use of specific clients, is granted by now Publishers Inc for users registered with the Copyright Clearance Center (CCC). The 'services' for users can be found on the internet at: www.copyright.com

For those organizations that have been granted a photocopy license, a separate system of payment has been arranged. Authorization does not extend to other kinds of copying, such as that for general distribution, for advertising or promotional purposes, for creating new collective works, or for resale. In the rest of the world: Permission to photocopy must be obtained from the copyright owner. Please apply to now Publishers Inc., PO Box 1024, Hanover, MA 02339, USA; Tel. +1-781-871-0245; www.nowpublishers.com; sales@nowpublishers.com

now Publishers Inc. has an exclusive license to publish this material worldwide. Permission to use this content must be obtained from the copyright license holder. Please apply to now Publishers, PO Box 179, 2600 AD Delft, The Netherlands, www.nowpublishers.com; e-mail: sales@nowpublishers.com

**Foundations and Trends® in
Electronic Design Automation**
Volume 3 Issue 1, 2008
Editorial Board

Editor-in-Chief:

Sharad Malik

Department of Electrical Engineering

Princeton University

Princeton, NJ 08544

Editors

Robert K. Brayton (UC Berkeley)

Raul Camposano (Synopsys)

K.T. Tim Cheng (UC Santa Barbara)

Jason Cong (UCLA)

Masahiro Fujita (University of Tokyo)

Georges Gielen (KU Leuven)

Tom Henzinger (EPFL)

Andrew Kahng (UC San Diego)

Andreas Kuehlmann (Cadence Berkeley Labs)

Ralph Otten (TU Eindhoven)

Joel Phillips (Cadence Berkeley Labs)

Jonathan Rose (University of Toronto)

Rob Rutenbar (CMU)

Alberto Sangiovanni-Vincentelli (UC Berkeley)

Leon Stok (IBM Research)

Editorial Scope

Foundations and Trends[®] in Electronic Design Automation will publish survey and tutorial articles in the following topics:

- System Level Design
- Behavioral Synthesis
- Logic Design
- Verification
- Test
- Physical Design
- Circuit Level Design
- Reconfigurable Systems
- Analog Design

Information for Librarians

Foundations and Trends[®] in Electronic Design Automation, 2008, Volume 3, 4 issues. ISSN paper version 1551-3939. ISSN online version 1551-3947. Also available as a combined paper and online subscription.

Applied Assertion-Based Verification: An Industry Perspective

Harry Foster

Mentor Graphics Corporation, Plano, Texas, USA
Harry.Foster@mentor.com

Abstract

A wealth of material has been published over the past 30 years specifically related to the theory and technical aspects of property languages and assertion-based techniques. However, as any field of study matures, it becomes necessary to determine if the theories, algorithms, and concepts have grown beyond the bounds of research to become an integral solution to a problem in industry. To understand any solution, it is necessary to understand the problem. For example, debugging, on average, has grown to consume more than 60% of today's ASIC and SoC verification effort. Clearly, this is a topic the industry must address, and some organizations have done just that. Those that have adopted an assertion-based verification (ABV) methodology have seen a significant reduction in simulation debugging time (as much as 50% [1, 47]) due to improved observability. Furthermore, organizations that have embraced an ABV methodology are able to take advantage of more advanced verification techniques, such as formal property checking, thus improving their overall verification quality and results. This paper examines the application of ABV in today's electronic design industry to address specific challenges of poor observability and controllability during the

verification process. Statistics illustrating successful application of both low-level and high-level assertions are presented. While the process of writing assertions is fairly well understood by those skilled in the art — the process of creating higher-level assertion-based IP that must communicate with other components in a contemporary transaction-level modeling (TLM) simulation environment, is not. Hence, this paper provides a set of steps (in a tutorial fashion) for creating assertion-based IP.

Contents

1	Introduction	1
1.1	What is an Assertion?	2
1.2	Controllability and Observability	3
1.3	Assertion Stakeholders	5
1.4	Assertions Within a Flow	7
1.5	Assertion-Based Verification Discussion Preview	8
2	Industry Adoption of Assertion-Based Verification	9
2.1	The Productivity Gap	9
2.2	Industry ABV Case Studies	13
2.3	Enabling Advanced Functional Verification	14
3	Assertion Languages and Libraries	17
3.1	Reasoning About Behavior	17
3.2	Assertion Language Standards	22
3.3	Property Specification Language (PSL)	24
3.4	SystemVerilog Assertions (SVA)	33
3.5	Open Verification Library (OVL)	37
4	Creating Assertion Intellectual Property	41
4.1	Guiding Principles	42
4.2	Development Steps	43
4.3	Assertion-Based IP Architecture	44

5 Bus Protocol Assertion Intellectual Property Example	61
5.1 Block Diagram Interface Description	61
5.2 Overview Description	61
5.3 Natural Language Properties	66
5.4 Formalize Properties	66
5.5 Functional Coverage Properties	75
5.6 Encapsulate Properties	77
6 Assertion-Based Verification Within a Flow	79
6.1 ABV and Coverage Closure	79
6.2 Assertions and Measuring Proof Progress	82
6.3 Addressing Formal ABV Complexity	84
6.4 Summary	92
References	95

1

Introduction

Ensuring functional correctness on RTL designs continues to pose one of the greatest challenges for today's ASIC and SoC design teams. Very few project managers would disagree with this statement. In fact, an often cited 2004 industry study by Collett International Research revealed that 35% of the total ASIC development effort was spent in verification [10]. In 2008, a Far West Research study (in conjunction with Mentor Graphics) indicated the verification effort has risen to 46% of the total ASIC development effort [20]. Furthermore, these industry studies reveal that debugging is the fastest growing component of verification, and that it consumes 60% of the total verification effort. Unfortunately, with the increase in verification effort, the industry has not experienced a measurable increase in quality of results. For example, a Collett International Research study that focused on design closure indicated that only 29% of projects developing ASICs were able to achieve first silicon success. To make matters worse, the industry is witnessing increasing pressure to shorten the overall ASIC and SoC development cycle. Clearly, new design and verification techniques, combined with a focus on maturing functional verification process

2 Introduction

capabilities within an organization (and the industry as a whole) are required.

Assertion-based verification (ABV), although certainly not an end-all to the verification challenge, does directly address today's debugging problem, while providing an integration path for more advanced forms of verification into the design flow (such as formal property checking). This paper provides a survey of today's ABV landscape, ranging from industry case studies to today's assertion language standardization efforts, to emerging challenges and research opportunities.

In addition, this paper directly addresses industry process issues of developing assertion-based IP by introducing a systematic set of planning and development steps. A detailed bus protocol example is provided, which draws together the various concepts introduced throughout the text while demonstrating an effective process for developing assertion and assertion-based verification IP.

1.1 What is an Assertion?

Alan Turing made the following observation over 50 years ago: "How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily flows [46]." In essence, this view is at the heart of ABV.

Informally, an assertion is a statement of design intent that can be used to specify design behavior. Assertions may specify internal implementation design behaviors (such as a specific FIFO structure) or external specification design behavior (such as a bus protocol or even higher-level, end-to-end behavior that spans multiple design blocks). One key characteristic of assertions is that they allow the user to specify *what* the design is supposed to do at a high level of abstraction, without having to describe the details of *how* the design intent is to be implemented. Thus, this abstract view of the design intent is ideal for the verification process — whether we are specifying high-level

requirements or lower-level implementation behaviors — by means of assertions.

Properties versus assertions: The recent flurry of interest in assertion-based techniques has prompted considerable published research and industry articles on the subject. Often, the authors of these publications interchange the terms *property* and *assertion*, which leads to confusion. For our discussion, a property is informally defined as follows:

Property — A statement of design intent.

For instance, the statement, *grant0 and grant1 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 the verification process. 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 from the verification tool 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 specification of design intent and *assert* that the property holds on all traces produced by a verification tool.

For our discussion, an assertion is informally defined as follows:

Assertion — An implementation of a property that is evaluated or executed by a tool to validate design intent.

Assertions are used as targets during the verification process (for example, a checker for simulation or a proof obligation for formal) to help us identify and isolate unexpected behavior.

1.2 Controllability and Observability

Fundamental to the discussion of ABV is understanding the concepts of *controllability* and *observability* [13, 19]. Informally, controllability refers to the ability to influence or activate an embedded finite state machine, structure, or specific line of code within the design by stimulating various input ports. Note that, while in theory a simulation

4 Introduction

testbench has high controllability of the design model's input ports during verification, it can have very low controllability of an internal structure within the model. Observability, in contrast, refers to the ability to observe the effects of a specific internal finite state machine, structure, or stimulated line of code. Thus, a testbench generally has limited observability if it only observes the external ports of the design model (because the internal signals and structures are often indirectly hidden from the testbench).

To identify a design error using a simulation testbench approach, the following conditions must hold:

1. The testbench must generate proper input stimulus to activate a design error.
2. The testbench must generate proper input stimulus to propagate all effects resulting from the design error to an output port.

It is possible, however, to set up a condition where the input stimulus activates a design error that does not propagate to an observable output port. In these cases, the first condition cited above applies; however, the second condition is absent, as illustrated in Figure 1.1.

Embedding assertions in the design model increases observability. In this way, the verification environment no longer depends on generating input stimulus to propagate a design error to an observable port. Thus,

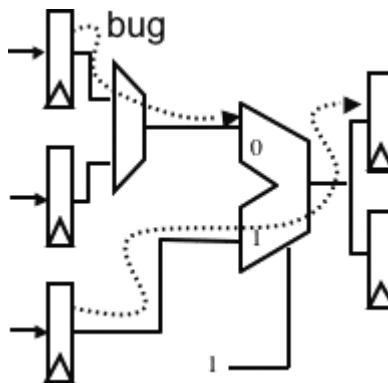


Fig. 1.1 Poor observability and controllability misses bugs.

any improper or unexpected behavior can be caught closer to the source of the design error, in terms of both time and location. Thus resulting in an overall reduction of debugging time.

While embedded assertions help solve the observability challenge in simulation, they do not help with the controllability challenge. However, the existence of assertions within the flow does open up the possibility for utilizing formal property checking to target critical or high-value assertions, thus addressing the controllability challenge.

1.3 Assertion Stakeholders

Assertions added at any level of hierarchy (or abstractions) clearly benefit verification by reducing debugging time while clarifying design intent. Certainly multiple stakeholders within the design and verification process can 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. They can be called implementation assertions. This adoption trend 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.

Although an architect can 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 that specify correct interface behavior between units and between blocks. The verification engineer also contributes by developing black-box, end-to-end assertions across design components.

Although it is easy to identify the various stakeholders and roles they play in the assertion development process, these stakeholders

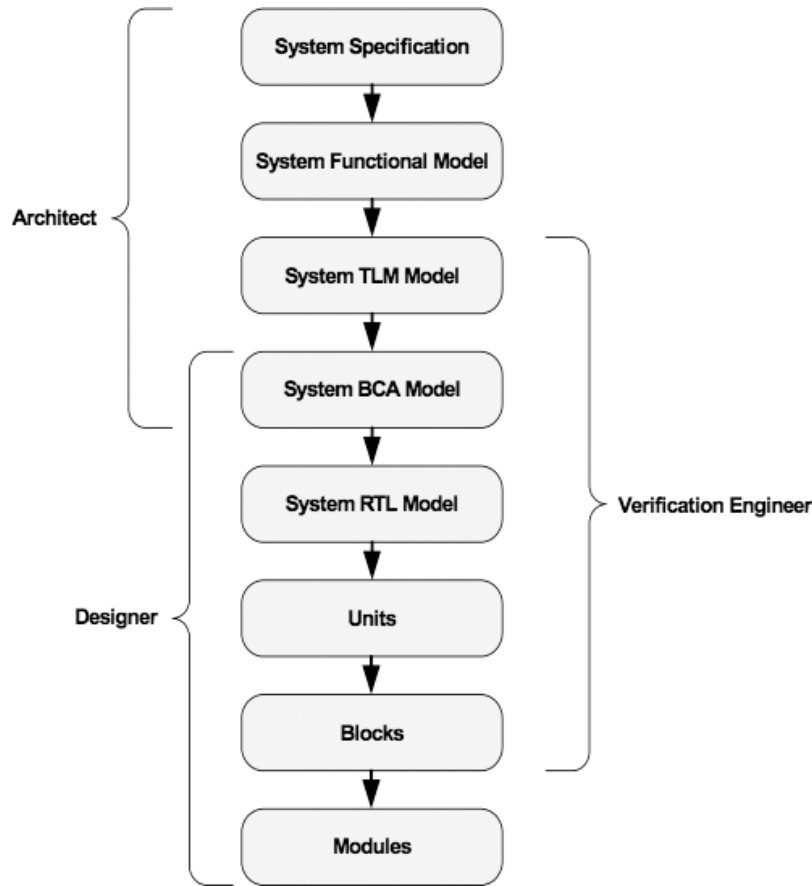


Fig. 1.2 Stakeholders and levels of abstraction.

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. This problem can be addressed by introducing a systematic process in which a natural language list of properties is created prior to coding the assertion-based IP, as discussed later in this paper.

1.4 Assertions Within a Flow

When applying assertion-based techniques, it is important to first understand where they can be effectively applied. For example, in addition to the verification flow stakeholders, Figure 1.2 also illustrates a typical design refinement process.

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 [39]. 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.

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

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

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 of the levels of abstraction illustrated in Figure 1.2. For example, in an untimed TLM, one concurrent transaction might overlap with a different transaction. That is, it can begin and end in the middle execution 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 [14, 15, 16]. Today, successful assertion specification in an industry setting occurs at and below the BCA abstraction level, illustrated in Figure 1.2. Later in this paper,

techniques are presented for developing assertion-based IP that can be reused in a transaction-level testbench using existing standards by introducing what is essentially an abstraction converter between a timed RTL model and untimed transaction environment.

1.5 ABV Discussion Preview

The remainder of this paper is organized as follows: In Section 2, industry motivations for adopting ABV are presented. Included are a number of successful case studies. Section 3 is focused on the recent emergence of assertion language and library standards. As a foundation for the assertion language discussion, a basic introduction to temporal logic and extended regular expressions is presented, followed by a brief introduction to IEEE Property Specification Language (PSL), SystemVerilog Assertions (SVA), and the Open Verification Library (OVL). Section 4 provides a tutorial for developing assertion-based IP with a focus on contemporary simulation environments found in industry. Section 5 integrates the concepts presented in all the previous sections into a demonstration on how to create assertion-based IP for a simple nonpipelined bus protocol example. Finally, Section 6 provides a summary discussion.

2

Industry Adoption of ABV

Prior to discussing the benefits that assertion-based techniques can contribute to today's verification flow, it is important to understand the verification challenges that industry currently faces. These challenges are real and highlighted by various industry studies that emphasize today's productivity gaps. Part of today's productivity gap is exacerbated by the failure of a large portion of industry to move beyond traditional verification techniques (such as directed test and code coverage). These approaches are unable to keep pace with increased design sizes, concurrency, and complexity.

2.1 The Productivity Gap

In the early 1990's, the design community moved design up a level of abstraction from gate-level to RT-level. Evidence of this shift is shown in Figure 2.1, with the increase in the curve representing the ability to design [10, 30]. Yet, even with today's synthesis breakthroughs in design productivity, designing and synthesizing RTL entirely from scratch cannot keep pace with what it is possible to fabricate (within a project's schedule). Hence, solutions, such as the adoption of third-party IP, will be necessary to increase design productivity in the future.

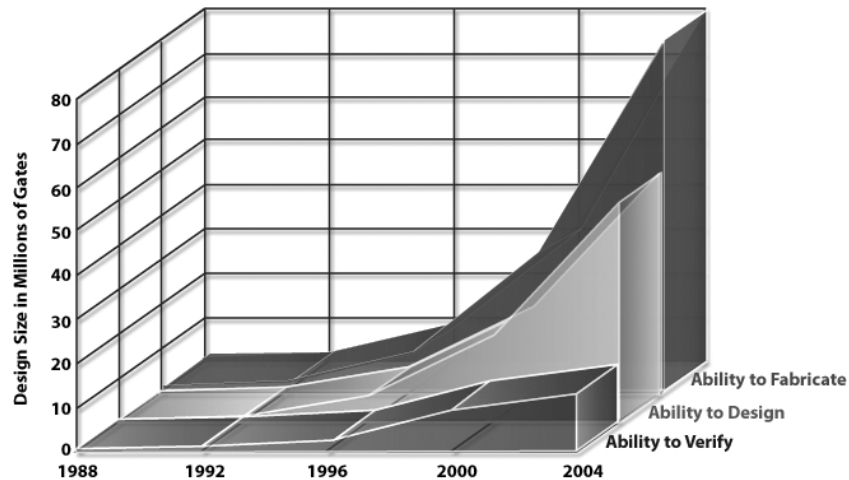


Fig. 2.1 Productivity gap.

A further examination of Figure 2.1, draws a disparity between what can be designed and what it is possible to verify. Yet in many respects, the data in Figure 2.1 seems to defy reality. Project teams actually do successfully verify complex chips today, which is obvious from the myriad new electronic products available, although industry studies indicate that only one third of today's project achieved their desired schedule [10]. Today's verification gap is not due to a lack of innovation in verification technology. What differentiates a successful team from an unsuccessful team is process and adoption of new verification methods, as revealed in the list of processes adopted by projects achieving first silicon success [10, 20]. Unsuccessful teams tend to approach development in an *ad hoc* fashion, while successful teams employ a more mature level of methodology that is systematic.

In the early to mid-1990's, the industry began to witness a partitioning of project responsibilities into separate design and verification teams. This was partially driven by the specialized software skill set that was necessary to build the required complex verification environments. Since the mid-1990's, the growth of verification effort has been alarming. A 2007 IC/ASIC functional verification study conducted by

Far West Research [20] found that the makeup of today's design teams, on average, consisted of:

- 51% design engineers;
- 35% verification engineers;
- 14% others.

Yet 46% of the design engineers were involved in verification in some capacity during the course of their project. Hence, an overwhelming amount of resources are involved in the verification process.

2.1.1 The Concurrency Challenge

Clearly, there are many forces at play today that contribute to the gap between what we can design and what we have time to verify. In addition, there are forces that contribute to a gap between what we can fabricate (silicon capacity) and what we have time to design. What is alarming in the data behind the Collett verification gap studies is that the state-of-the-art best practices applied by approximately 90% of the study participants were directed test and code coverage.

To illustrate the verification productivity gap concern using a simplified example, imagine trying to verify a specific property for an automobile using only a directed-test approach. Our property states that *the car will not blow a fuse under any normal operation*. Using a directed test approach, a verification engineer would:

- Think of a scenario to test.
- Design a directed test to cover this scenario.
- Apply the test and observe the results.

Then the verification engineer would repeat this process for every scenario imaginable.

For example:

- *Scenario 1*: Accelerate the car to 37 mph, pop in a CD, and turn on the windshield wipers.
- *Result 1*: Fuse did not blow.



Fig. 2.2 Real world concurrency challenge.

A few weeks later...

- *Scenario 714*: Accelerate the car to 48mph, turn on the radio, and roll down the window.
- *Result 714*: Fuse blew!!!!

The problem with this approach is that it becomes unmanageable as the number of scenarios increase, often due to concurrency. Imagine the challenge in trying to describe the scenario illustrated in Figure 2.2, which holds similarities to the multi-channel packet-based design illustrated in Figure 2.3.

Clearly, functional verification flows that consist entirely of directed test and code coverage approaches (circa 1990 best practices) struggle to keep pace with today's verification challenges. In fact, there have been many helpful verification technologies that have emerged in the past 15 years (such as constrained-random, coverage-driven simulation; formal property checking; and ABV) that have demonstrated value by increasing productivity and achieving higher quality, as indicated both by those achieving first silicon success in the Collett and Far West studies.

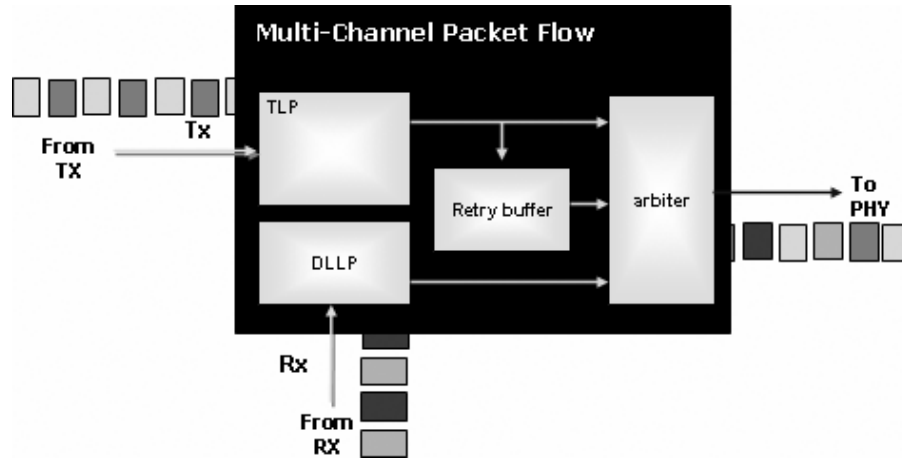


Fig. 2.3 Real design verification concurrency challenge.

2.2 Industry ABV Case Studies

Although the industry shows significant interest in assertion-based techniques, predominantly driven by the emergence of assertion language standards in the 2004 timeframe, many companies first started researching and applying assertion-based techniques in the mid- to late 1990s, including the following prominent companies [22]:

- Cyrix Corporation.
- Digital Equipment Corporation.
- Hewlett-Packard Company.
- IBM Corporation.
- Intel Corporation.
- Motorola, Inc.
- Silicon Graphics, Inc.
- Sun Microsystems. Inc.

Designers from these companies describe their success with methodologies that incorporate assertions as follows:

- 34% of all bugs were found by assertions on DEC Alpha 21164 project [32].

- 17% of all bugs were found by assertions on Cyrix M3(p1) project [22, 33].
- 25% of all bugs were found by assertions on DEC Alpha 21264 project — The DEC 21264 Microprocessor [45].
- 25% of all bugs were found by assertions on Cyrix M3(p2) project [22, 34].
- 85% of all bugs were found using OVL-style assertions at HP [21].

Of particular note is the DAC 1996 DEC Alpha 21164 case study. The title of this study is, “I’m done simulating, now what?” From a verification perspective, it appeared that the project team had completed the validation of the design since the bug rate had leveled off. However, after adding assertions (as well as functional coverage properties), one third of the total bugs identified throughout the verification process were detected by an assertion. In fact, simulation tests that had previously passed prior to adding assertions were suddenly failing, due to improved observability that identified previously undetected bugs.

In addition to successfully detecting design failures, assertions significantly reduced debugging time, as measured and described in the following case studies:

- 50% reduction in simulation debugging time published by IBM [1].
- 50% reduction in simulation debugging and 85% reduction in formal debugging time published by Sun Microsystems [47].

From these case studies, a common theme emerges: when design and verification engineers use assertions as a part of the methodology, they are able to detect a significant percentage of design failures while reducing debugging time as a result of improved observability.

2.3 Enabling Advanced Functional Verification

Part of the verification productivity gap previously illustrated in Figure 2.1 can be attributed to the failure of many organizations within the industry to mature their verification processes beyond a

purely directed-test and code-coverage methodology. As previously noted, a directed-test verification methodology will not keep pace with the increased size, complexity, and concurrency challenges that have resulted from Moore’s Law. To reduce the need for handcrafting tests, new advanced functional verification techniques are required. While the evidence shows that assertions help solve the observability challenge, it also shows that they are used to address the controllability challenge by enabling the adoption of constrained-random, coverage-driven simulation techniques and formal property checking.

For example, a fundamental component of a constrained-random methodology is the creation of a functional coverage model [42]. This coverage model consists of a set of coverage properties that can be described using today’s assertion language standards. Without a functional coverage model, it is impractical (if not impossible) to know exactly what functionality has been exercised by the constrained-random-generated stimulus. However, once coverage properties have been defined, it is possible to measure (i.e., identify) functionality that has not been exercised, which enables re-biasing the simulation constraints in an attempt to focus the generated stimulus towards the unexplored functionality. Therefore, coverage properties enable measuring controllability issues and then taking appropriate actions to improve coverage.

Similarly, assertion and assumption properties form the foundation of today’s formal property checking methodology — thus addressing traditional dynamic functional verification controllability challenges through exhaustive static-search techniques. In terms of industry adoption of advanced functional verification techniques, the 2008 Far West Research [20] study indicated that approximately 19% of the industry has adopted formal property checking (i.e., model checking [9]), while approximately 41% of the industry has adopted constrained-random, coverage-driven techniques. Part of the challenge with the adoption of assertion-based techniques in the industry is due to the expertise required in grasping temporal logic and declarative forms of specification. These concepts, which are discussed in the next section, form the foundation of today’s assertion languages and are foreign to many of today’s engineers trained in electronics.

3

Assertion Languages and Libraries

This section provides an introduction to the assertion languages and library standards that are used in industry. It begins by briefly introducing temporal logic, which forms the basis for modern property specification languages. Building on this foundation, it then presents a historical perspective for today's assertion language and library standards.

3.1 Reasoning About Behavior

Logic, whose origins can be traced back to ancient Greek philosophers, is a branch of philosophy (and today mathematics) concerned with reasoning about behavior. In a classical logic system, we state a proposition, and then deduce (or infer) whether a given model satisfies our proposition, as illustrated in Figure 3.1 [12].

For instance, consider a set of propositions from a classic example:

- *The moon is a satellite of the earth.*
- *The moon is rising (now).*

If we let the universe be our model, then using classical logic, we can check whether our set of propositions hold for the given model.

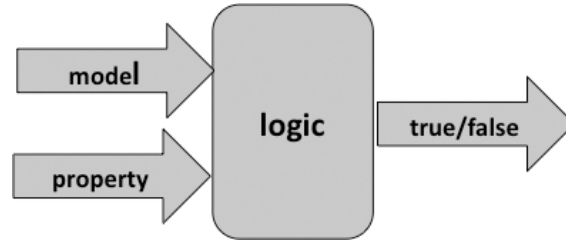


Fig. 3.1 Classical logic system.

Classical logic is good for describing static situations. However, classical logic is unsuitable for describing dynamic behavior (i.e., situations involving time). Returning to the previous example, it would not be possible to express the following proposition since it involves time:

- *The moon will rise again and again.*

Note that an interest in functional verification of hardware systems requires the use of logic that is expressive enough to describe properties of reactive systems. For a reactive system, components of the system concurrently maintain ongoing interaction with their environment, as well as other concurrent components of the system. Hence, the next section discusses a more expressive logic that involves time.

3.1.1 Temporal Logic

This section builds a foundation of understanding by introducing a few basic concepts of propositional temporal logic. The advantage of using temporal logic to specify properties of reactive systems is that it enables reasoning about these systems in a very simple way. That is, temporal logic eliminates the need to explicitly specify the time relationships between system events, which are represented as Boolean formulas that describe states of the design. For example, instead of writing the property expression involving time explicitly, such as:

$$\forall t.!(\text{grant1}(t) \ \& \ \text{grant2}(t))$$

which specifies for all values of time t , grant1 and grant2 are mutually exclusive, simply write in a temporal property language, such as PSL:

always !(grant1 & grant2)

which states that grant1 and grant2 should not hold at the same time.

In temporal logic, we define a computational path π as an infinite sequence of states

$$\pi = s_0, s_1, s_2, s_3, \dots$$

that represents a forward progression of time and a succession of states, s_i . When verifying a temporal assertion, assume that a point in time or given state along the path has a unique future or successor state (e.g., $s_0 \rightarrow s_1$), in which case, the assertion is verified on a given path or trace of execution (e.g., a single simulation trace). Thus, each possible computational path of a system is considered separately. This form of logic is called linear time, temporal logic, and LTL is one example [43]. As an alternative, it is possible to assume that each point in time or given state along a path may split into multiple future or successor states (e.g., $s_0 \rightarrow s_1$ and $s_0 \rightarrow s_5$). In that case, all computational paths are considered concurrently, which is usually represented as a tree of infinite computational paths. We refer to this form of logic as branching time, temporal logic, and CTL (Computational Tree Logic) is one example [8].

CTL and LTL play essential roles in formal hardware verification for branching time, temporal logic and linear time, and temporal logic. To show how these two types of logic differ, this paper first introduces CTL* [18]. CTL* contains operators for reasoning about paths of computation, such as the path formula operators G (always), F (eventually), U (until) and X (next) and operators for reasoning on branching paths of execution, i.e., the state formula operators A (for all paths of execution) and E (for any path of execution). In addition to these quantifiers, any Boolean compositions of CTL* formulas are also CTL* formulas.

For any temporal formulas p and q , as illustrated in Figure 3.2, the temporal formula Gp specifies that the temporal formula p holds for all states of a path π (or simply, p always holds). The temporal formula Fp

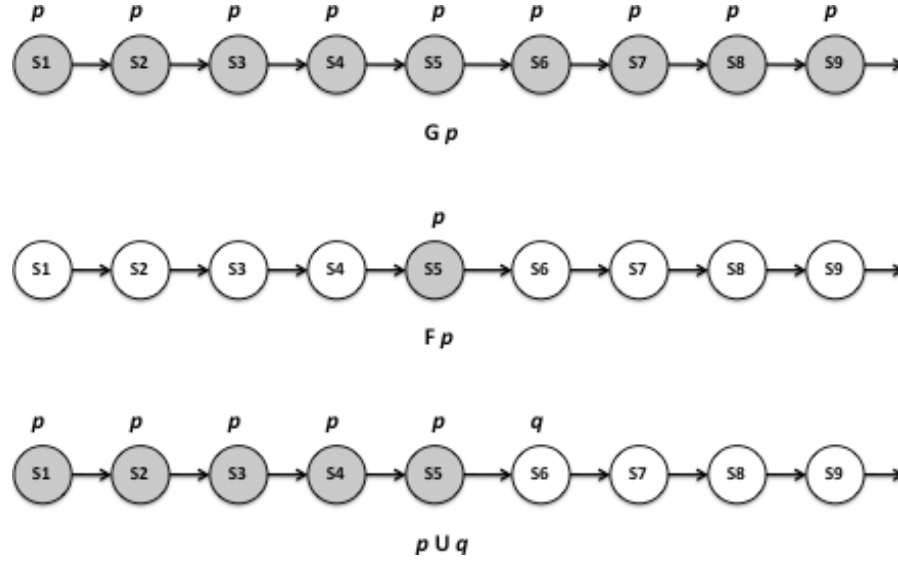


Fig. 3.2 Temporal formula path operator semantics.

specifies that p holds for some future state of a path π . The temporal formula $p U q$ specifies that the temporal formula p holds in all states of a path π until q holds in some future state of π .

The temporal formula $A f$ specifies that for all paths π starting from the current state, f holds. The temporal formula $E f$ means that there is a path π starting from the current state for which f holds.

As seen in the previous paragraphs, CTL* can be separated into state formulas (involving A and E) and path formulas (involving G , F , X , and U). Any atomic proposition p (or Boolean expression) over state formulas is by definition a state formula. In addition, existential or universal quantification over path formulas (e.g., $E f$, where E is the existential quantifier and f is a path formula) is also by definition a state formula.

If f and g are state formulas, then $X f$, $F f$, $G f$, and $f U g$ are path formulas. Note that the F operator can be thought of as an alias for the unary form of the until operator (e.g., $F p = \text{true} U p$), and that G and F are dual (i.e., $G p$ is equivalent to $\neg F \neg p$, where $\neg$ is the negation symbol). The rationale behind the first alias is that eventually (F)

is equivalent to waiting vacuously until p is valid. And the rationale behind the second formula is that saying “ p always valid” is equivalent to saying that “it is not the case that $\neg p$ will be valid in the future.” Similarly, it is not difficult to show that A and E are dual (i.e., Ap is equivalent to $\neg E \neg p$).

Note that in CTL* we do not make any restriction on the order in which temporal and branching operators appear in a valid formula. As a result, FGf and AGf are valid CTL* formulas. The first formula (FGf) states that eventually our proposition f will be valid forever. The second formula (AGf) states that for all paths starting from the current state, our proposition f will always be true.

Now that we have presented CTL*, we can restrict this logic to CTL and LTL.

- A CTL formula is a CTL* formula beginning with a branch quantifier (A and E), restricting that temporal operators (F, G, U, and X) be preceded by branch quantifiers.

For example, the formula AGp is a valid CTL formula, but FGp is not.

- An LTL formula is the subset of CTL* formulas obtained by simply restricting the valid formulas to path formulas.

For example, the formula FGp is a valid LTL formula, but EGp is not.

If we consider an implicit universal quantifier for all paths (i.e., A) in front of an LTL formula, we can see that certain behaviors, such as $A(GFp)$, cannot be represented by CTL, although it is a valid LTL formula. Similarly, $AG(EFp)$ is a valid CTL formula, but not a valid LTL formula. While the first formula says that for all states of all paths, eventually p holds (a fairness constraint), the second formula says that for all branches of all states, at least in one of the paths, eventually p holds.

Note that neither LTL nor CTL can express a property that involves counting, such as:

- p is asserted in every even cycle.

The next section introduces extended regular expressions, which express behavior that involves counting.

The semantics of CTL*, CTL and LTL are not thoroughly discussed in this section, but a short introduction on the subject is given to support the remainder of this paper. Refer to [35] for a more complete definition of these logics' semantics and complexities.

3.1.2 Extended Regular Expressions

Regular expressions provide a convenient way to define sets of computations (i.e., a temporal pattern used to match various sequences of states). Extended regular expressions are regular expressions extended with conjunctions and negation. Hence, the computational path defined by an extended regular expression can be combined to form the building blocks used to specify hardware design assertions. Regular expressions can express counting modulo n type behavior through the use of the $*$ operator. Hence, regular expressions specify properties that cannot be described by either CTL or LTL, such as:

$$\{\backslash\text{true};p\}^*$$

which is a PSL sequential extended regular expression example that states that p is asserted in every even cycle.

Not all properties can be expressed with extended regular expressions. For example, the property eventually p holds forever cannot be expressed using extended regular expressions, yet this property can be expressed in LTL (i.e., $\text{FG}p$).

3.2 Assertion Language Standards

Assertions are certainly not a new phenomenon in either software programming or hardware description languages. For example, languages such as Java and VHDL have contained simple assertion constructs for years — thus providing a convenient mechanism for identifying a class of bugs that can be identified by checking a Boolean condition.

Today's assertion languages, such as the IEEE Std 1850TM-2005 Standard for Property Specification Language [29] and the assertion language contained in the IEEE Std 1800TM-2005 SystemVerilog

Verification and Hardware Description Language [28], not only allow the user to specify Boolean conditions, they also their relation over time using temporal logic and a generalized form of regular expressions.

The foundation for today's assertion language standards is built on the works of Pnueli (linear time logic (LTL) [43]), and Clarke and Emerson (computation tree logic (CTL) [8]). Furthermore, the works of Vardi and Wolper provided significant contributions in that they helped improve expressiveness of LTL through the use of regular sequences of Boolean events [49, 48, 50]. Extending the expressiveness of CTL was later demonstrated by Iwashita and Nakata [31].

In the early 1990's, researchers at the IBM Haifa Research Laboratory developed the temporal language Sugar, which was a syntactic sugaring of CTL. The goal was to simplify the specification process for the RuleBase model checker. To improve usability and expressiveness, regular expressions were added to the language in the mid-1990's [5]. By the late 1990's, IBM had expanded its use of the Sugar assertion language for simulation [1].

With a similar motive, researchers at Intel Strategic CAD Labs developed the ForSpec property specification language, whose underlying logic is the ForSpec Temporal Logic (FTL) [4], which is based on LTL. Their decision to base FTL on LTL was driven by a desire to combine formal verification and dynamic validation (i.e., simulation) techniques, which is only possible in a very limited fashion. Furthermore, experience had demonstrated that mainstream verification engineers generally find branching time unintuitive — particularly since they are familiar with dynamic validation, which is inherently linear.

In 2000, both Sugar and ForSpec, in addition to the temporal property languages CBV from Motorola and Temporal *e* from Verisity, were donated to Accellera Formal Verification Technical Committee (FVTC) as candidate languages for standardization. The process within the committee was to establish a set of requirements for an assertion language and select a single language from four candidates. The final selection would form the basis for the new standard, which then would undergo modification and enhancements dictated by the language requirements identified by the committee. For example, one of the