Verification Methodology Manual for SystemVerilog
When I co-authored the original edition of the Reuse Methodology Manual for System-on-Chip Designs (RMM) nearly a decade ago, designers were facing a crisis. Shrinking silicon geometry had increased system-on-chip (SoC) capacity well into the millions of gates, but development teams simply didn't have the time or resources to design so much logic while meeting product schedules. At that time, design reuse was emerging as the best way to resolve this dilemma. The RMM was written to provide an extensive set of rules, guidelines, and best practices for developing reusable IP that could be quickly and easily integrated into SoC designs.

IP-reuse-based SoC design methodology is now a fully accepted industry practice, and I am proud that the three editions of the RMM have helped to guide this evolution. It is now time for a book providing similar guidance for verification methodology and verification reuse. As many studies have shown, verification has emerged as the biggest portion of SoC development, consuming the most time and resources on most projects. The practices that sufficed for small designs-hand-written directed tests with minimal coverage metrics—are woefully insufficient in the SoC world.

I am pleased to introduce the Verification Methodology Manual for SystemVerilog, a book that will revolutionize the practices of verification engineers much as the RMM led designers to a better methodology with more predictable results. It encompasses all the latest techniques, including constrained-random stimulus generation, coverage-driven verification, assertion-based verification, formal analysis, and system-level verification in an open, well-defined methodology. It introduces and illustrates these techniques with examples from SystemVerilog, the industry standard linking RTL design, testbenches, assertions, and coverage together in a coherent and comprehensive language.
This book is not a theoretical exercise; it is based upon many years of verification experience from the authors, their colleagues, and their customers. It is practical and usable for SoC teams looking to greatly reduce the pain of verification while significantly increasing their chances of first-silicon success. It is my hope that the Verification Methodology Manual for SystemVerilog will be an essential reference guide for a whole new generation of SoC projects.

Pierre Bricaud
Co-Author of Reuse Methodology Manual for System-on-Chip Designs
Synopsys, Inc.
CONTENTS

Foreword v

Preface xiii

How this Book is Structured xi
How to Read this Book xiii
For More Information xv
Acknowledgements xvi

CHAPTER 1 Introduction 1

Verification Productivity 2
Increasing Productivity 3
Verification Components 4
Interface-Based Design 4
Design for Verification 6
The Benefit of Assertions 7
Methodology Implementation 8
Methodology Adoption 8
Guidelines 11
Basic Coding Guidelines 12
Definition of Terms 13
## CHAPTER 2 Verification Planning

- Planning Process .................................................. 18
  - Functional Verification Requirements .................. 18
  - Verification Environment Requirements .............. 22
  - Verification Implementation Plan ...................... 29
- Response Checking ................................................. 31
  - Embedded Monitors ............................................ 32
  - Assertions ...................................................... 33
  - Accuracy ....................................................... 36
  - Scoreboarding ................................................. 38
  - Reference Model .............................................. 39
  - Offline Checking ............................................... 40
- Summary .......................................................... 41

## CHAPTER 3 Assertions

- Specifying Assertions ........................................... 44
  - Assertion Language Primer ................................. 46
- Assertions on Internal DUT Signals ....................... 50
- Assertions on External Interfaces .......................... 59
- Assertion Coding Guidelines .................................. 63
  - Coverage Properties ......................................... 72
- Reusable Assertion-Based Checkers ....................... 77
  - Simple Checkers ................................................ 78
  - Assertion-Based Verification IP ........................... 86
  - Architecture of Assertion-Based IP ..................... 90
  - Documentation and Release Items ....................... 99
- Qualification of Assertions ................................... 100
- Summary .......................................................... 102

## CHAPTER 4 Testbench Infrastructure

- Testbench Architecture ....................................... 104
  - Signal Layer .................................................... 107
  - Command Layer ............................................... 116
  - Functional Layer .............................................. 118
  - Scenario Layer ............................................... 122
  - Test Layer ..................................................... 123
- Simulation Control .............................................. 124
CHAPTER 5   Stimulus And Response

Generating Stimulus ......................... 211
   Random Stimulus  ......................... 213
   OOP Primer: Factory Pattern ............... 217
   Directed Stimulus ......................... 219
   Generating Exceptions .................... 221
   Embedded Stimulus ....................... 226

Controlling Random Generation ............ 227
   Atomic Generation ....................... 231
   Scenario Generation ..................... 232
   Multi-Stream Generation ................. 236
   State-Dependent Generation .............. 238
   Which Type of Generator to Use? .......... 244

Self-Checking Structures ................. 246
   Scoreboarding  ......................... 249
   Integration with the Transactors ......... 253
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Coverage-Driven Verification</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td>Coverage Metrics</td>
<td>260</td>
</tr>
<tr>
<td></td>
<td>Coverage Models</td>
<td>261</td>
</tr>
<tr>
<td></td>
<td>Structural Coverage Modeling</td>
<td>262</td>
</tr>
<tr>
<td></td>
<td>Functional Coverage Modeling</td>
<td>263</td>
</tr>
<tr>
<td></td>
<td>Functional Coverage Analysis</td>
<td>265</td>
</tr>
<tr>
<td></td>
<td>Coverage Grading</td>
<td>266</td>
</tr>
<tr>
<td></td>
<td>Functional Coverage Implementation</td>
<td>266</td>
</tr>
<tr>
<td></td>
<td>Coverage Groups</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>Coverage Properties</td>
<td>276</td>
</tr>
<tr>
<td></td>
<td>Feedback Mechanisms</td>
<td>277</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>280</td>
</tr>
<tr>
<td>7</td>
<td>Assertions for Formal Tools</td>
<td>281</td>
</tr>
<tr>
<td></td>
<td>Model Checking and Assertions</td>
<td>282</td>
</tr>
<tr>
<td></td>
<td>Assertions on Data</td>
<td>292</td>
</tr>
<tr>
<td></td>
<td>Without Local Variables</td>
<td>293</td>
</tr>
<tr>
<td></td>
<td>With Local Variables</td>
<td>297</td>
</tr>
<tr>
<td></td>
<td>Compatibility with Formal Tools</td>
<td>302</td>
</tr>
<tr>
<td></td>
<td>Summary</td>
<td>303</td>
</tr>
<tr>
<td>8</td>
<td>System-Level Verification</td>
<td>305</td>
</tr>
<tr>
<td></td>
<td>Extensible Verification Components</td>
<td>306</td>
</tr>
<tr>
<td></td>
<td>XVC Architecture</td>
<td>306</td>
</tr>
<tr>
<td></td>
<td>Implementing XVCs</td>
<td>309</td>
</tr>
<tr>
<td></td>
<td>Implementing Actions</td>
<td>311</td>
</tr>
<tr>
<td></td>
<td>XVC Manager</td>
<td>316</td>
</tr>
<tr>
<td></td>
<td>Predefined XVC Manager</td>
<td>317</td>
</tr>
<tr>
<td></td>
<td>System-Level Verification Environments</td>
<td>319</td>
</tr>
<tr>
<td></td>
<td>Block Interconnect Infrastructure Verification</td>
<td>323</td>
</tr>
<tr>
<td></td>
<td>Basic Integration Verification</td>
<td>326</td>
</tr>
<tr>
<td></td>
<td>Low-Level System Functional Verification</td>
<td>328</td>
</tr>
<tr>
<td></td>
<td>System Validation Verification</td>
<td>329</td>
</tr>
<tr>
<td></td>
<td>Verifying Transaction-Level Models</td>
<td>332</td>
</tr>
</tbody>
</table>
CHAPTER 9  Processor Integration Verification 343

Software Test Environments 343
  Basic Software Integration Verification 345
  Full System Verification Environment 346

Structure of Software Tests 349

Test Actions 354
  Compilation Process 359
  Running Tests 361
  Bootstrap 363

Summary 364

APPENDIX A  VMM Standard Library Specification 365

vmm_env 365
vmm_log 368
  vmm_log_msg 378
  vmm_log_format 379
  vmm_log_callbacks 381
vmm_data 383
vmm_channel 387
vmm_broadcast 397
vmm_scheduler 401
  vmm_scheduler_election 404
vmm_notify 405
  vmm_notification 409
vmm_xactor 411
  vmm_xactor_callbacks 415
vmm_atomic_gen 415
  <class_name>_atomic_gen_callbacks 418
vmm_scenario_gen 418
  <class_name>_scenario 421
  <class_name>_atomic_scenario 424
  <class_name>_scenario_election 425
<class_name>_scenario_gen_callbacks .......... 426

APPENDIX B  VMM Checker Library 429

OVL-Equivalent Checkers (SVL) ................. 429
Advanced Checkers ............................... 434

APPENDIX C  XVC Standard Library Specification 439

xvc_manager ........................................ 439
xvc_xactor ......................................... 440
xvc_action ......................................... 442
vmm_xvc_manager .................................. 444
  Notifications .................................... 444
  File Structure ................................... 445
  Commands ........................................ 447

APPENDIX D  Software Test Framework 459

Basic Types ........................................ 459
System Descriptor .................................. 460
  Peripheral Descriptor ............................ 460
  Interrupt Descriptor ............................ 463
  DMA Channel Descriptor ......................... 464
Test Actions ........................................ 465
Low-Level Services ................................. 470
  Cache Lockdown .................................. 474
  Interrupt Controller ............................ 475
  Software-XVC Connectivity ....................... 478

Index .............................................. 481

About the Authors ................................ 503
When VHDL first came out as an IEEE standard, it was thought to be sufficient to model hardware designs. Reality proved to be a little different. Because it did not have a predefined four-state logic type, each simulator and model vendor had to create its own—and incompatible—logic type. This situation prompted the quick creation of a group to create a standard multi-valued logic package for VHDL that culminated with the 1164 standard. With such a package, models became interoperable and simulators could be optimized to perform well-defined operations.

The authors of this book hope to create a similar standard for verification components within the SystemVerilog language. The infrastructure elements specified in the appendices can form the basis of a standard verification interface. If model vendors use it to build their verification components, they will be immediately interoperable. If simulator vendors optimize their implementation of the standard functions, the runtime performances can be improved.

HOW THIS BOOK IS STRUCTURED

The book is composed of chapters and appendices. The chapters describe guidelines that must or should be followed when implementing the verification methodology. The appendices specify application-generic support elements to help in the implementation process.

Chapter 3 provides guidelines for writing assertions. Its companion Appendix B specifies a set of predefined checkers that can be used in lieu of writing new assertions.
Preface

Chapter 4 describes the components of a verification environment and how to implement them. Its companion Appendix A specifies a set of base and utility classes that are used to implement the generic functionality required by all environments and components.

Chapter 5 describes how to provide stimulus to the design under verification and how it can be constrained to create interesting conditions. The generator classes specified in Appendix A help to rapidly create VMM-compliant generator components.

Chapter 6 describes how to use qualitative metrics to drive the verification process and using a constrainable random verification environment built using the guidelines presented in the previous chapters to efficiently implement it.

Chapter 7 describes how assertions can be used with formal technology. Only a subset of the checkers described in Appendix B can be used within this context.

Chapter 8 describes how the principles presented in the previous chapters can be leveraged for system-level verification. Its companion Appendix C specifies a command language and extensible component infrastructure to implement block and system-level verification environments.

Chapter 9 describes how the integration of a general-purpose programmable processor in a system can be verified using a set of predefined C functions described in Appendix D.

The support infrastructure is specified in appendices A through D by describing the interface and functionality of each element. No implementation is provided. It is up to each vendor to provide a suitable implementation. This gives the opportunity to EDA or IP vendors to optimize the implementation of the infrastructure for their particular platform. It also eliminates the risk that unintended side effects of a particular "reference" implementation might be interpreted as expected behavior. The code for the interface specifications is available at the companion Web site:

http://vmm-sv.org

Note that the methodology can be followed without using the support elements specified in the appendices. Any functionally equivalent set of elements, providing similar functionality, would work. However, using a different set of support elements will likely diminish the interoperability of verification components and environments written using different support infrastructures.
HOW TO READ THIS BOOK

This book is not designed as a textbook that can be read and applied linearly. Although the authors have made their best effort to present the material in a logical order, it will be often difficult to appreciate the importance or wisdom of some elements of the methodology without a grasp of the overall picture. Unfortunately, it is not possible to draw an overall picture without first building the various elements used to construct it.

The chicken-and-egg paradox is inherent to describing methodologies. A methodology is about taking steps today to make life easier in some future. A successful methodology will help reduce the overall cost of a project through investments at earlier stages that will provide greater returns later on. In a practical description of a methodology, it is difficult to justify some of the initial costs as their future benefit is not immediately apparent. Similarly, describing the future benefits is not possible without describing the elements that, when put together, will create these benefits.

A reader unfamiliar with an equivalent methodology would typically require two readings of the entire book. A first reading will help form the overall picture of the methodology, how the various elements fit together and the benefits that can be realized. A second reading will help appreciate its detailed implementation process and supporting library.

Although everyone will benefit from reading the entire book, there are sections that are more relevant to specific verification tasks. Designers must read Chapter 3. They should also read Chapter 7 if they intend to use formal technology to verify their design. Verification leaders and project managers should read Chapters 2 and 6. Verification engineers responsible for the implementation and maintenance of the verification environment must read Chapters 4 and 5 and should read Chapter 8. Verification IP developers should read Chapters 4 and 8. Verification engineers responsible for implementing testcases should read the first half of Chapter 5. If they are also responsible for implementing functional coverage points, they should read the second half of Chapter 6. Embedded software verification engineers should read Chapter 9.
FOR MORE INFORMATION

At the time of writing, SystemVerilog was in the process of being ratified as an IEEE standard. In addition to several books already—or to be—published, more information about SystemVerilog can be obtained from:

http://www.eda.org/sv
http://www.eda.org/sv-ieee1800

This book assumes the reader has experience with the entire SystemVerilog language. It is not designed as an introductory or training text to the verification or assertion constructs. The following books, listed in alphabetical order, can be used to gain the necessary experience and knowledge of the language constructs:

Janick Bergeron, "Writing Testbenches Using SystemVerilog", Springer


Chris Spear and Arturo Salz, "SystemVerilog for Verification", Springer

In this book, code examples are provided as extracts that focus on the various points they are designed to illustrate. It does not contain a full example of the methodology application within its page. Such an example would consume several tens of pages filled with SystemVerilog code. It would be difficult to navigate, would become obsolete as improvements to the methodology are made and impossible to actually simulate. Pointers to several complete examples and the complete code that includes the various examples can be found at the companion Web site:

http://vmm-sv.org

The companion Web site will also contain an errata for the latest edition of the book. It may also publish additional guidelines as the methodology evolves and is expanded. These additional guidelines will be included in future editions. Discussions on the use or interpretation of the methodology and suggestions for improvement are carried in the forums at:

http://verificationguild.com

ACKNOWLEDGEMENTS

The authors would like to thank Holger Keding for his contribution to Chapter 8. They are also grateful for the thoughtful reviews and challenging comments from Pierre Aulagnier, Oliver Bell, Michael Benjamin, Jonathan Bradford, Craig Deaton,
Jeff DelChiaro, Geoff Hall, Wolfgang Ecker, Rémi Francard, Christian Glaßner, Olivier Haller, Takashi Kambe, Masamichi Kawarabayashi, Michael Keating, Dave Matt, Aditya Mukherjee, Seiichi Nishio, Zenji Oka, Michael Röder, Kostas Siomalas, Stefan Sojka, Jason Sprott, STARC IP Verification SWG (Masahiro Furuya, Hiroyuki Fukuyama, Kohkichi Hashimoto, Masanori Imai, Masaharu Kimura, Hiroshi Koguchi, Hirohisa Kotegawa, Youichiro Kumazaki, Yoshikazu Mori, Tadayuki Nakamura, Sanae Saitou, Masayuki Shono, Tsuneo Toba, Hideo Washimi, Takeru Yonaga), Rob Swan, Yoshio Takamine, Gary Vrckovnik and Frazer Worley. The book also benefited from the loving attention of Kyle Smith, the technical editor.

Many others have contributed to making this book and its content a reality. Alphabetically, they are: Jay Alphey, Tom Anderson, Tom Borgstrom, Dan Brook, Dan Coley, Tom Fitzpatrick, Mike Glasscock, John Goodenough, Badri Gopalan, David Gwilt, Tim Holden, Ghassan Khoory, FrameMaker, Mehdi Mohtashemi, Phil Moorby, Dave Rich, Spencer Saunders, David Smith, Michael Smith, Manoj Kumar Thottasseri and the VCS and Magellan implementation team.
CHAPTER 1  INTRODUCTION

In the process of design and verification, experience shows that it is the latter task that dominates time scales. This book defines a methodology that helps minimize the time necessary to meet the verification requirements. It also takes the opportunity offered by the definition of a methodology to also define standards that will enable the creation of interoperable verification environments and components.

Using interoperable environments and components is essential in reducing the effort required to verify a complete product. A consistent usage model is present in all verification environments. System-level environments are able to leverage the components, self-checking structure and coverage from block-level environments. Formal tools are able to share the same properties used by simulation. Verification IP is able to meet the requirement of verifying the interface block as well as the system-level functionality that resides behind it.

The methodology described in this book defines standards for specifying reusable properties that are efficient to simulate and that can be formally verified. It defines standards for creating transaction and data descriptors to facilitate their constrainable random generation while maintaining a flexible directed capability. This methodology standardizes how bus-functional models, monitors and transactors are designed to provide stimulus and checking functions that are relevant from block to system. Furthermore, this methodology sets standards for the integration of the various components into a verification environment so they can be easily combined, controlled and later extricated to leverage in different environments. The book also defines standards for implementing coverage models and software verification environments. These standards, when put together in a coherent methodology, help reduce the effort required to verify a design.
The methodology described in this book could be implemented using a different language or a different set of standards. But interoperability is maximized when the same language and the same set of standards are used. The classes and associated guidelines specified in this book can be freely used by anyone, users and EDA vendors alike. The objective is to create a vibrant SystemVerilog verification ecosystem that speaks a common language, uses a common approach and creates highly interoperable verification environments and components.

**VERIFICATION PRODUCTIVITY**

The progress of a verification project is measured by the number of functional features that are confirmed as functionally correct. Therefore, verification productivity is a measure of how many such features are confirmed as functionally correct over a period of time, including the time necessary to debug and fix any functional errors in the same features. The greater the productivity, the faster a high-quality product can be manufactured. This measure of productivity is not necessarily correlated to the amount of code written in the same time period, nor is it correlated to the runtime performance of the simulations used to confirm functional correctness. It is possible to achieve a higher verification productivity while writing less code and running more concurrent simulations.

Historically, verification methodologies have evolved alongside the design abstraction and kept pace with the complexities of the designs being implemented. When design was done at the mask level, verification was accomplished by simulating transistor models. When design transitioned to standard cells, verification transitioned to gate-level digital simulations. When design took advantage of the simulation language to introduce logic synthesis, verification evolved to transaction-level testbenches using bus-functional models. Throughout these evolutionary steps, the approach to verification has not fundamentally changed: Individual design features are verified using individual testcases crafted to exercise the targeted feature.

However, the traditional individual testcase approach does not scale to handle today’s largest multi-million gate designs. A project with one thousand separate features to verify would require over one calendar year to complete with the help of a team of 10 verification engineers able—on average—to write, debug and maintain one testcase every three days for the entire duration of the project. That effort requires an unusually large team of unusually productive engineers. Such large projects require a different approach.
Increasing Productivity

The methodology presented in this book improves the productivity of a verification project through four different mechanisms: assertions, abstraction, automation and reuse.

When using assertions to identify defects on interfaces or in runtime assumptions, errors are reported close in space and time to their ultimate cause. Otherwise, the consequence of the error may have been detected after several clock cycles if and when it reached a monitored output and checked against expectations. Some classes of errors produce symptoms that are easy to detect at the boundary of the design—a missing packet for example. However, some classes of errors have symptoms that are not so obvious—for example, an arbitration error that can be recovered from, only producing a small reduction in throughput for a certain class of service. Assertions create monitors at critical points in the design without having to create separate testbenches where these points would be externally visible.

Verifying at an increasing the level of abstraction is simply continuing the past historical trend. But unlike historical increases in abstraction, this one need not be accompanied by an equivalent increase in the design abstraction. It is still necessary to verify low-level implementation and physical details. Once low-levels of functionality are verified, verification can proceed at higher levels using a layered testbench architecture. The layering of transactors to form successive layers of abstraction is also used to break away from the monolithic bus-functional model that makes it difficult to introduce additional or combinations of protocol layers.

The design-specific nature of the tests and the response-checking mechanism makes general purpose automation of the verification process impossible. True automation would produce the exact same testcases that would be written manually. But random stimulus can emulate automation: Left to its own devices, a properly-designed random source will eventually generate the desired stimulus. Random stimulus will also create conditions that may not have been foreseen as significant. When random stimulus fails to produced the required stimulus, or when the required stimulus is unlikely to be produced by an unbiased random stimulus source, constraints can be added to the random stimulus to increase the probability (sometimes to 100%) of generating the required stimulus. Due to the random nature of the stimulus, it is necessary to use a coverage mechanism to identify which testcases have been pseudo-automatically produced so far. This coverage metrics measure the progress and productivity of the verification process. Verification requirements that were automatically met are quickly identify, allowing the effort to be concentrated on those that remain to be met.
Reusing code avoids having to duplicate its functionality. Reuse is not limited to reusing code across projects. First-order reuse occurs when the same verification environment is reused across multiple testcases on the same project. By reusing code as much as possible, a feature can be verified using just a few lines of additional code. Ultimately, testcases should become simple reconfigurations of highly reusable verification components forming a design-specific verification environment or platform. Note that this book is not about a reuse methodology. Reuse is only a means, not an end.

**VERIFICATION COMPONENTS**

As stated previously, first-order reuse occurs when a design-specific verification environment is reused across testcases for that design. Second-order reuse occurs when some components of the design-specific verification environment are reused in a system-level environment. Third-order reuse occurs when those same components are reused across different verification environments for different designs. For all of these reuse opportunities to be realized, verification components have to be properly designed.

For verification components to be reusable, they must be functionally correct and they must be configurable to meet the needs of the environments and testcases built on top of them. The term *configurable* in this context refers to the ability of the verification component to exhibit the required functionality to adequately exercise the design or system under verification. A configurable verification component can be used to drive an interface in a block-level environment. Later, the same component can be used, without modification, in a system-level verification environment. A verification component must thus meet the different stimulus and monitoring requirements of a block-level environment and a system-level environment. This book describes methodologies to build, then leverage, verification components to reduce the verification effort and increase reusability.

**Interface-Based Design**

Nowadays, designs have external interfaces and on-chip buses which, in all likelihood, implement industry-standard protocols such as the AMBA™ Protocol Family, USB or Utopia. The benefits of standardized external interfaces and on-chip buses are well understood and include availability of design IP, reusability of existing validation components during design development and ease of understanding by engineers during development.
Verification Components

Early in the design process, functional partitioning takes place to make the detailed design phase both manageable and suitable for execution by an engineering team. As shown in Figure 1-1, this procedure introduces many new internal interfaces into the design.

![Figure 1-1. Interface-Based Design](image)

Being internal, design engineers are free to implement these interfaces. There can often be as many different implementations of interfaces as interfaces themselves. To validate a partitioned design, verification components are required to stimulate each internal interface from the perspective of each agent on that interface. The number of verification components required for a partitioned design is therefore potentially proportional to the number of interfaces, which in itself grows exponentially with the number of partitions. For example, verifying the partitioned design shown in Figure 1-1—with three internal interfaces and four external interfaces—requires 13 different verification components, as illustrated in Figure 1-2.

![Figure 1-2. Verification Environments for Partitioned Design](image)

An interface-based design methodology should have internal interfaces well specified early in the design process, aim to minimize the number of unique interfaces and leverage a library of common verification components. This approach will enable designers to concentrate on the value-add of the design while meeting the performance goals of the various interfaces.
Introduction

DESIGN FOR VERIFICATION

Design for verification is a response to the problems encountered when verifying the current (and future) complex microelectronics devices. Like design for synthesis and design for test, it requires a change in how designs are specified, captured and implemented.

Design for synthesis methodologies introduced, along with RTL-based specifications, specific hardware design language (HDL) coding styles and required synchronous designs. These restrictions allowed the use of a set of tools supporting the methodology—logic synthesis, cycle-based simulation, static timing analysis and equivalence checking—which contributed to increasing the overall design productivity.

Design for test separated testing for structural defects from verifying the functional aspects of the devices. These methodologies imposed further restrictions on the designs—synchronous interfaces, no clock gating, no latches and exclusive bus drivers—but also came with additional tool support. These methodological restrictions, coupled with the tools that supported them, helped improve the controllability and observability of internal structural failures to yield enormous gains in device reliability at a much lower cost.

Design for verification involves the designer in the verification process as early as possible, even before—and especially during—the design process itself. Small upfront investments by designers can reap a substantial reduction in the effort and time required to verify a design. Design for verification includes providing a way for the designer to express his or her intent concisely and naturally as part of the design process so that it can be objectively verified. To that end, SystemVerilog provides assertions to check the behavior and assumptions of the design and interface signals. It also provides specific language constructs—such as the always_comb block—to remove ambiguity and further specify the intent of the implementation code.

Design for verification also encourages designers to make architectural and design decisions that minimize the verification costs of the system. For example, a write-only and a read-only register can share the same address, but making every writable register readable helps in verifying that they can be written to correctly. Similarly, minimizing the number of unique internal interfaces and using industry standard external interfaces and on-chip buses, as mentioned in “Interface-Based Design” on page 4, helps minimize the number of verification components that must be created. The ability to preset large counters, bypass computation paths or force exception status bits can also greatly ease the verification process. These decisions may require the addition of nonfunctional features in the design.
Design for Verification

Design for verification elements and features would not be exercised during the normal operations of the design. Some, such assertions, are usually removed from the final design by the synthesis process. However, these elements and features can help in on-chip diagnostics and debugging. For example, assertions may be synthesized into an emulated version of the design and their failure indication routed to a status register where they can generate an interrupt should they fail.

**The Benefit of Assertions**

The three main sources of functional flaws in taped-out designs are design errors, specification errors and errors in reused modules and IP (either internal errors or incorrect usage). Most of these errors are due to ambiguous or changing specifications or unwritten or unverified assumptions on the behavior of surrounding blocks.

When creating the RTL implementation of a design, the designer often makes assumptions on the behavior of the surrounding designs and on internal synchronization. These assumptions are usually extraneous to the specification, unwritten and not verified during simulation. Any change in the behavior of the surrounding designs or functional errors in internal synchronization may violate these assumptions, which leads to failures. The symptoms of these failures may not be apparent until much later in the simulation—if at all—when the data affected by the failure reaches an observed output. These undocumented assumptions make the detection and the identification of the cause of a failure difficult and time consuming.

Designers should state such assumptions using assertions and insert them into the RTL code where these assumptions are used. A violation of these assumptions would cause an assertion failure near the point in space and time of the ultimate cause of the failure. This approach makes debugging the design that much easier.

A similar situation exists when reusing an existing module or IP block. If the assumptions on the usage of the module are not precisely stated and verified, errors may be difficult to identify due to the black-box nature of reused designs. Assertions can play an important role in specifying the usage rules of the module.

Temporal constructs, like those available in SystemVerilog, provide an efficient means for system architects to complement design specifications with non-ambiguous and executable statements in the form of properties that, when asserted, precisely express the intent or requirement of the specification. Such assertions reduce ambiguity and thus the chance of misinterpretation. Since properties are a different, more abstract description of the required behavior than the RTL specification of the design’s implementation, they increase the likelihood of detecting a design error.
Introduction

during simulation. Moreover, formal and hybrid (a combination of formal engines and simulation) tools can prove that the design does not violate some properties under any legal input stimulus.

Finally, properties can be used to describe interesting stimulus and states that should be covered during verification. These properties are not used to detect failures, but to detect the occurrence of some important condition. They specify corner cases created by the chosen implementation architecture that may not have been obvious based on the functional specification alone. A designer can thus contribute to the verification plan of a design by including coverage properties in the RTL design. Similarly, several compliance statements of standard protocols can be implemented using coverage properties.

**METHODOLOGY IMPLEMENTATION**

The methodology presented in this book is quite extensive. It contains several different—but interrelated—facets and elements. The increase in productivity that can be obtained by this methodology comes from its breath and depth. It is not a methodology designed to be tidily presented in a 30-minute conference paper or three-page journal article on a toy example. It is designed to be scalable and applicable to real-life designs and systems.

The goal of the methodology is to obtain the maximum level of confidence in the quality of a design in a given amount of time and engineering resources. To accomplish this goal, it uses assertions, functional abstraction, automation through randomization and reuse techniques *all at the same time*. The guidelines presented in the subsequent chapters are designed to implement a verification process that combines all of these techniques to maximum effect.

It may be difficult to appreciate the usefulness of a particular set of guidelines without knowing the overall methodology implementation. But it is equally difficult to effectively implement a methodology without detailed guidelines to construct its basic elements. It may thus be beneficial to read this book twice: the first time to learn the overall methodology; the second time to learn its implementation details.

**Methodology Adoption**

It is not necessary to adopt all the elements of the methodology presented in the following chapters. Obviously, maximum productivity is achieved when all of the synergies between the elements of the methodology are realized. But real projects, with real people and schedules, may not be able to afford the ramp-up time necessary
for a wholesale adoption. Individual elements of the methodology can still be adopted and provide incremental benefits to a project.

Many design teams already use assertions to detect errors in the design or interface signals. Many other books have already been written on their benefit. Adopting the methodology elements presented in Chapter 3 will accelerate their correct application, development and deployment. The same chapter also describes how to construct reusable assertion-based checkers that can be used without knowing the underlying assertion language, thus minimizing the cost of introducing assertions into an existing design methodology.

The message service, embodied in the `vmm_log` class described in “Message Service” on page 134, is the easiest one to adopt. It can immediately replace the message routines or packages that most teams develop for themselves. Its adoption requires no change in methodology and provides additional functionality at no cost. But unlike traditional message packages, it allows messages from verification components reused from different sources to be consistently displayed and controlled, without modifying the reused component itself.

Formalizing the simulation steps described in “Simulation Control” on page 124, as embodied in the `vmm_env` base class, is the next natural adoption step. All simulations have to perform the same overall sequence of steps to successful completion. Instead of creating an arbitrary synchronization and sequencing scheme with each new verification environment, the `vmm_env` base class helps to formalize the execution steps in a consistent and maintainable fashion. Different block-level environments will be easier to combine to create system-level environments if they use a similar control mechanism. Formalizing the simulation steps allows different tests to intervene at the appropriate time, without ever violating the simulation sequence.

Modeling transactions using transaction descriptors and building all data models upon the `vmm_data` base class, as described in “Data and Transactions” on page 140, creates a uniform stimulus creation mechanism described in “Controlling Random Generation” on page 227. Whether transactions or data, all stimulus is created the same way. Both can be easily randomized and constrained using identical mechanisms. Different tests can apply different constraints without having to rewrite the random generator. Fully directed or partially directed transactions can also be easily created.

Once transactions and data are modeled as objects, interfacing transactors using `vmm_channels` as described in “Transaction-Level Interfaces” on page 171 and decoupling transactor functionality according to protocol and abstraction layers
Introduction

comes next. Using these channels allows the creation of finer-grain plug-and-play transactors that can be reused across verification environments within the same project. Having a well-defined transaction-level interface mechanism enables the creation of transactors operating at higher level of abstraction without having to be associated with a physical level interface like traditional bus-functional models. It also enables the construction of verification environments along layers that can be built top-down—first on a transaction-level model of the DUT then adapted to a RTL model—or bottom-up—first verifying low-level operations then on to more complex and abstract functions.

A major step must be taken to adopt factory-patterned generators described in “Random Stimulus” on page 213 and callback methods in transactors as described in “Transactors” on page 161. But they offer the ability to create tests with fewer lines of codes in a single file, without modifying—and potentially breaking—code that is known to work. Tests can be written without affecting any of the already-written tests. And because tests can be written with so few lines to target a specific function of the device under test, it becomes cost effective to implement a true coverage-driven verification methodology. As a secondary benefits, generators and transactors that can meet the unpredictable needs of different tests, will be able to meet the needs of different verification environments or projects, making them truly reusable.

If the nature of the corner cases of the system depends on the synchronization of concurrent stimulus on multiple interfaces, adopting the extensible verification component (XVC) approach described in Chapter 8 becomes a good idea. Once the interesting stimulus parameters for an interface are known and implemented, it provides a natural command-based interface for writing system-level tests unlikely to occur spontaneously in a purely random environment. And should more stimulus parameters or patterns be required, they can be easily added to the environment without requiring modifications to the existing, working environment. It also creates an easy-to-learn test specification mechanism that can be used without being familiar with the details of the entire methodology or its implementation.

Formal tools are very effective in finding hard-to-identify or hard-to-reach corner-case bugs on complex control-dominated design blocks, such as arbiters, bus protocol controllers, instruction schedulers, pipeline controls, and so on. The RTL implementation of these structures is compared against a description of their expected behavior using assertions. When writing assertions that can be formally proven as well as simulated, the guidelines described in Chapter 7 become pertinent in addition to those in Chapter 3.

If there is any software component to the project, verifying the hardware/software interaction will require the adoption of the techniques described in Chapter 9.
Guidelines

The purpose of this book is not to extol the virtues of SystemVerilog and the verification methodology it can support. Rather, like its predecessor the Reuse Methodology Manual, this book is focused on providing clear guidelines to help the reader make the most effective use of SystemVerilog and implement a productive verification methodology. The book does not claim that its methodology is the only way to use SystemVerilog for verification. It presents what the authors believe to be the best way.

Not all guidelines are created equal, and the guidelines in this book are classified according to their importance. More important guidelines should be adopted first, then eventually supported by a greater set of less important guidelines. However, it is important to recognize the synergies that exist among the guidelines presented in this book. Even if they are of lesser importance, adopting more of the guidelines will generally result in greater overall efficiency in the verification process.

Rules — A rule is a guideline that must be followed to implement the methodology. Not following a rule will jeopardize the productivity gains that are offered by other aspects of the methodology. SystemVerilog Verification Methodology Manual-compatibility (VMM-compatibility) requires adherence to all rules. VMM-compliance requires that all rules be followed.

Recommendations — A recommendation is a guideline that should be followed. In many cases, the detail of the guideline is not important—such as a naming convention—and can be customized. Adherence to all recommendations within a verification team or business unit is strongly recommended to ensure a consistent and portable implementation of the methodology.

Suggestions — Suggestions are recommendations that will make the life of a verification team easier. Like recommendations, the detailed implementation of a suggestion may not be important and may be customizable.

Alternatives — Alternatives provide different mechanisms for achieving similar results. Different alternatives may not be equally efficient or relevant and depend on the available verification components and the verification environment being constructed.

The guidelines in this book focus on the methodology, not the tools that support SystemVerilog or other aspects of this methodology. Additional guidelines may be required to optimize the methodology with a particular toolset.
Basic Coding Guidelines

There is no value in reiterating generic coding guidelines—such as rules for indentation and commenting—that can be found in a variety of sources. All previous coding guidelines applicable to Verilog will remain applicable to SystemVerilog. However, the presence of high-level and verification-centric constructs in SystemVerilog require that some additional basic coding guidelines be specified.

Recommendation 1-1 — Unique prefixes or suffixes should be used to identify the construct that implements user-defined types.

SystemVerilog has a rich set of user-definable types: interface, struct, union, class, enums. It may be difficult to know what construct is used to implement all user-defined types, and thus what are the allowed operations on objects of that type. Using a construct-specific prefix or suffix helps identify the underlying implementation.

Example 1-1. Using Unique Construct-Specific Suffixes

```
typedef enum ... states_e;
typedef struct ... header_s;
typedef union ... format_u;
class packet_c;
    ...
endclass: packet_c
interface mii_if;
    ...
endinterface: mii_if
```

Recommendation 1-2 — End tags should be used.

SystemVerilog supports end tags on all of the named constructs that create a scope. Because there can be several dozens or hundreds of lines within that scope, indentation alone is often insufficient to clearly identify the matching opening and closing markers. By using end tags, associating a closing marker with its corresponding marker is much easier.

Example 1-2. Using End Tags

```
function compute_fcs(...);
    ...
    if (...) begin: is_special
        ...
    end: is_special
    ...
endfunction: compute_fcs
```
Definition of Terms

Verification is about communication. A design must be unambiguously specified to be correctly implemented and verified against that specification. Verification often identifies miscommunication of intent among the various teams in a project. A book about verification must also be as unambiguous as possible. The following section defines the terminology used in this book. Other works may use the same terms to mean other things.

Assertion — A property that must hold true at all times. if statements and the entire verification environment can be considered assertions. But in this book, the term refers only to the behavior described using the property specification constructs.

Assertion-based verification — The systematic use of assertions to help identify design faults and specify assumptions on input signals.

Assertion coverage — A measure of how thoroughly an asserted property has been exercised. Does not imply any measure of functional intent. Part of code coverage.

Bus-functional model — A transactor with a physical-level interface.

Code coverage — A measure of the structural code constructs exercised during specific simulations. Includes several metrics, such as line coverage, path coverage, toggle coverage, expression coverage and assertion coverage.

Checker — A verification component that verifies the correctness of a protocol. Low-level checkers are usually implemented using assertions. Checkers may be combined with monitors.

Class property — A data member in a class type declaration.

Constrained-random — A modification of a verification environment, through additional constraints, to increase the likelihood that specific stimulus will be generated.

Coverage — A measure of progress of the verification process. Includes several coverage metrics, such as code coverage, functional coverage and FSM coverage.

Coverage property — A property that, when true, indicates that an interesting condition has occurred. The occurrence is recorded in a database for later analysis. Coverage properties can be used to implement functional coverage points.
Cross coverage — The combination of two or more coverage metrics to measure their relative occurrences. Cannot be used to combine heterogeneous coverage measurements, such as code coverage and functional coverage.

Data protection class property — A class data member implementing a protocol mechanism used to detect, and sometimes repair, errors in data or a transaction. FCS, CRC and HEC fields are examples of data protection class properties.

Design for test — Nonfunctional design requirements and activities to make structural testing of the manufactured design easier.

Design for verification — Nonfunctional design requirements and activities to make functional verification easier. Includes assertion-based verification.

Directed random — A synonym for constrained-random.

Directed testbench — Testbench specified using hand-crafted stimulus. Usually contains a hand-crafted description of the expected response as well. May include some random data for the irrelevant portions of the stimulus that do not affect the outcome of the feature targeted by the testcase. May be implemented on top of a random verification environment.

 Discriminant class property — A class data member, usually randomized, whose value determines the presence or absence of additional data representing different data or transaction formats.

FSM coverage — A measure of the visited states and transitions observed on a finite-state machine during specific simulations. Can be automatically extracted from the FSM implementation or independently specified.

 Formal verification — A mathematical comparison of an implementation against a specification or requirement to determine if the implementation can violate its specification or requirement.

Functional coverage — A measure of the testcases and interesting conditions that were observed as having been exercised on the design (e.g., corner cases, applied input scenarios, and so on.)

Functional coverage point — A specific testcase or interesting condition that must be observed as having been exercised on the design.

Generator — A proactive transactor that autonomously generates stimulus.