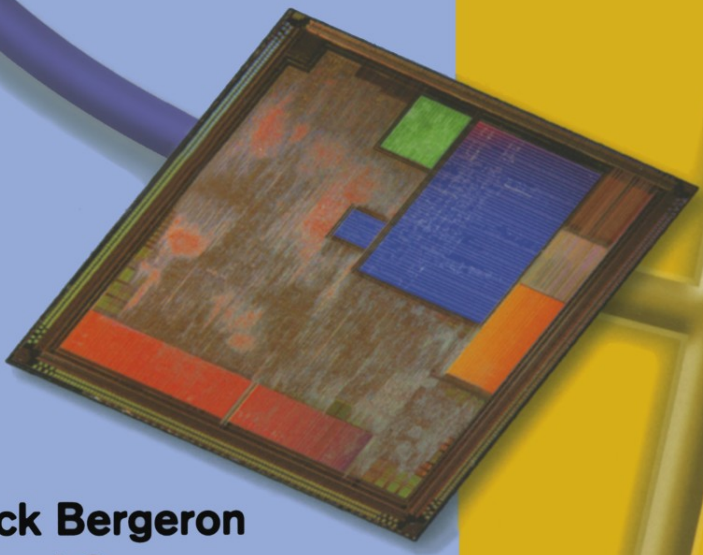
Verification Methodology Manual for SystemVerilog



Janick Bergeron
Eduard Cerny
Alan Hunter
Andrew Nightingale

Springer

Verification Methodology Manual for SystemVerilog

Verification Methodology Manual for SystemVerilog

by

Janick Bergeron
Eduard Cerny
Alan Hunter
Andrew Nightingale

Springer

Janick Bergeron, *Synopsys, Inc.*
Eduard Cerny, *Synopsys, Inc.*
Andrew Nightingale, *ARM, Ltd.*
Alan Hunter, *ARM, Ltd.*

Verification methodology manual for SystemVerilog / by Janick Bergeron … [et al.].
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-387-25538-5 (alk. paper)
ISBN-10: 0-387-25538-9 (alk. paper)
ISBN-10: 0-387-25556-7 (e-book)
1. Verilog (Computer hardware description language) 2. Integrated circuits--Verification.
I. Bergeron, Janick.

TK7885.7 V44 2005
621.39'2--dc22

2005051724

Cover: Die photograph of the ARM926EJ-S™ PrimeXsys™ Platform Development Chip © 2005 ARM Ltd.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1 SPIN 11055174

springeronline.com

FOREWORD

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

PREFACE

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.

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

Ben Cohen, Srinivasan Venkataramanan and Ajeetha Kumari, "*SystemVerilog Assertions Handbook"*, VhdlCohen Publishing

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, Tadahiko Nakamura, Sanae Saitou, Masayuki Shono, Tsuneo Toba, Hideaki 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.

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.

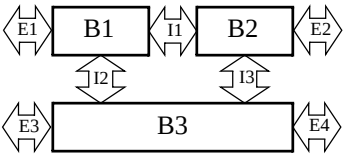


Figure 1-1. Interface-Based Design

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 designed shown in Figure 1-1—with three internal interfaces and four external interfaces—requires 13 different verification components, as illustrated in Figure 1-2.

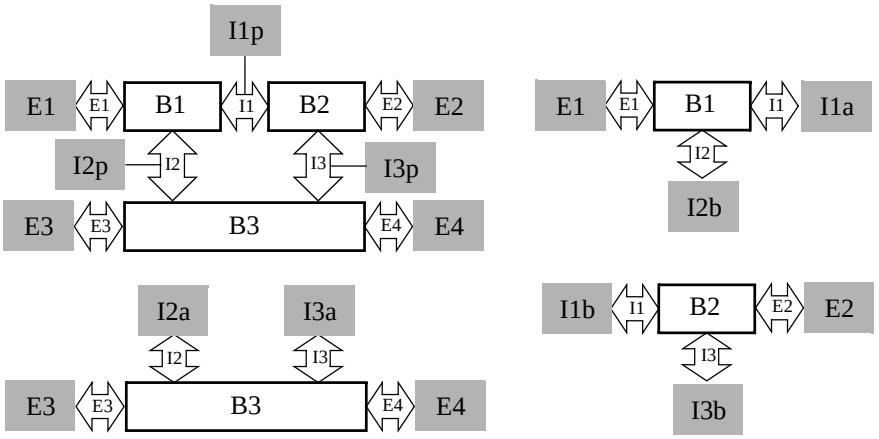


Figure 1-2. Verification Environments for Partitioned Design

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.

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 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

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_channel*s as described in "Transaction-Level Interfaces" on page 171 and decoupling transactor functionality according to protocol and abstraction layers

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*.

Monitor — A *reactive* or *passive transactor* that autonomously reports observed data or transactions. May includes a *checker* or equivalent checking functionality for the observed protocol, but not the data or transactions transported by the protocol.

Passive transactor — A *transactor* that strictly observes the design under verification or lower-level transactors. It has no control of the timing and initiation of transactions. May include *functional coverage points.* See *reactive transactor* and *proactive transactor.*

Proactive transactor — A *transactor* that provides stimulus to the design under verification or a lower-level transactor and is under full control of the timing and initiation of transactions. See *reactive transactor.*

Property — A specification of functional behavior using a sequence of Boolean expressions. Properties can be used to specify events, conditions that must always hold true or functional coverage points.

Random environment — Synonym of *verification environment.*

Reactive transactor — A *transactor* that provides stimulus to the design under verification or a lower-level transactor but has no control over the initiation of transactions. It may have limited control over the timing of certain elements of transaction initiated by the DUT or lower-level transactor, such as the insertion of wait states. See *proactive transactor* and *passive transactor.*

Scenario — A sequence of random or directed stimulus that is particularly interesting to the device under test. A scenario is unlikely to be spontaneously generated with individually constrained-random stimulus. Multiple scenarios are applied to the device under test during a single simulation.

Simulation — A run, with a specific seed and set of source files, of a model and associated testbench, resulting in a specific set of output messages and coverage metrics.

Stimulus — Input to the design under verification. Includes, but is not limited to, configuration information, transactions, instructions, exceptions and injected protocol errors.

Structural coverage — A synonym for *code coverage*.

System — A design composed of independently verified sub-designs. A system may go on to become an independently verified design in a super-system.

Test — A synonym for *testcase*.

Testcase — A requirement of the functional verification process. Usually corresponds to an interesting feature or corner condition of the design that must be verified.

Testbench — A complete verification environment applying *stimulus* and checking the response of a design to implement one or more *testcases*. A *testcase* can be verified using a *directed testbench* or *constrained-random* testbench with *functional coverage*.

Testing — The process to determine that a physical device was correctly manufactured without defects, according to a specific implementation.

Transaction — An operation on an interface. A transaction can be abstract and high-level—such as the reliable transmission of a TCP packet—or physical—such as a write cycle on a APB™ interconnect.

Transactor — A *verification component* of a *verification environment*. A transactor is a static object that autonomously generates, processes or monitors *transactions*. Bus-functional models are transactors—but transactors are not limited to bus-functional models. See *proactive transactor*, *reactive transactor* and *passive transactor*.

Validation — The process to determine that a specification meets its functional and market requirements.

Verification — The process to determine that an implementation meets its specification.

Verification component — A potentially reusable component of a *verification environment*. See *transactor*.

Verification environment — Verification automation, abstraction and response checking environment used to verify a specific design. Can be used to implement *constrained-random* simulations. Can be used to implement *directed testbenches*.

CHAPTER 2 VERIFICATION PLANNING

As stated in the previous chapter—and in several other published works—more effort is required to verify a design than to write the RTL code for it. As early as 1974, Brian Kernighan, creator of the C language, stated that "*Everyone knows debugging is twice as hard as writing a program in the first place*." A lot of effort goes into specifying the requirements of the design. Given that verification is a larger task, even more effort should go into specifying how to make sure the design is correct.

Every design team signs up for first-time success. No one plans for failures and multiple design iterations. But how is *first-time success* defined? How can resources be appropriately allocated to ensure critical functions of the design are not jeopardized without a definition of what functionality is critical? The *verification plan* is that specification. And that plan must be based on the intent of the design, not its implementation. Of course, corner cases created by the implementation, which are not apparent from the initial intent, have to be verified but that should be done once the initial intent-based verification plan has been completed.

This chapter will be of interest to verification lead engineers and project managers. It will help them define the project requirements, allocate resources, create a work schedule and track progress of the project of time. Examples in this chapter are based on OpenCore *Ethernet IP Core Specification*, Revision 1.19, November 27, 2002. This document can be found in the Examples section of the companion Web site:

```
http://vmm-sv.org
```

PLANNING PROCESS

The traditional approach of writing verification plans should be revised to take advantage of new verification technologies and methodologies. The new verification constructs in SystemVerilog offer new promises of productivity but only if the verification process is designed to take advantage of them. Individual computers, like individual language features, can improve the productivity of the person sitting in front of it. But taking advantage of the network, like the synergies that exist among those language features, can be achieved only by redesigning the way an entire business processes information; and it can dramatically improve overall efficiency.

The traditional verification planning process—when there is one—involves identifying testcases targeting a specific function of the design and describing a specific set of stimulus to apply to the design. Sometimes, the testcase may be self-checking and looks for specific symptoms of failures in the output stream observed from the design. Each testcase is then allocated to individual engineers for implementation. Other than low-level bus-functional models directly tied to the design's interface, little is reused between testcases. This approach to verification is similar to specifying a complete design block by block and hoping that, once put together, it will meet all requirements.

A design is specified in stages: first requirements, then architecture and finally detailed implementation. The verification planning process should follow similar steps. Each step may be implemented as separate cross-referenced documents, or by successive refinement of a single document.

Functional Verification Requirements

The purpose of defining functional verification requirements is to identify the verification requirements necessary for the design to fulfill the intended function. These requirements will form the basis from which the rest of the verification planning process will proceed. These requirements should be identified as early as possible in the project life cycle, ideally while the architectural design is being carried out. It should be part of a project's technical assessment reviews.

It is recommended that the requirements be identified and reviewed by a variety of stakeholders from both inside and outside the project team. The contributors should include experienced design, verification and software engineers so that the requirements are defined from a hardware and a software perspective. The reviews are designed to ensure that the identified functional verification requirements are complete.

Rule 2-1 — *A definition of what the design does shall be specified.*

Defining what the design does—what type of input patterns it can handle, what errors it can sustain—is part of the verification requirements. These requirements ensure that the design implements the intended functionality. These requirements are based on a functional specification document of the design agreed upon by the design and verification teams.

These requirements are outlined, separate from the functional specification document.

Example 2-1. Ethernet IP Core Verification Requirements

```
R3.1/14/0 Packets are limited to MAXFL bytes
R3.1/13/0 Does not append a CRC
R3.1/13/1 Appends a valid CRC
R4.2.3/1  Frames are transmitted
```

Rule 2-2 — *A definition of what the design must not do shall be specified.*

Defining what makes the behavior of the design incorrect is also part of the verification requirements. These requirements ensure that functional errors in the design will not go unnoticed. The section titled "Response Checking" on page 31 specifies guidelines on how to look for errors.

A functional specification document is concerned with specifying the intended behavior. That is captured by Rule 2-1. Verification is concerned with detecting errors. But it can only detect errors that it is looking for. The verification requirements must outline which errors to look for. There is an infinite number of ways something can go wrong. The verification requirements enumerate only those errors that are relevant and probable, given the functionality and architecture of the design.

Rule 2-3 — *Any functionality not covered by the verification process shall be defined.*

It is not possible to verify the behavior of the design under conditions it is not expected to experience in real life. The conditions considered to be outside the usage space of the design must be outlined to clearly delineate what the design is and is not expected to handle.

For example, a datacom design may be expected to automatically recover from a parity error, a complete failure of the input protocol or a loss of power. But a

processor may not be expected to recover from executing invalid instruction codes or a loss of program memory.

Example 2-2. Ethernet IP Core Verification Requirements

```
R3.1/9/0  Frames are lost only if attempt limit is
          reached
R4.2.3/2  Frames are transmitted in BD order
```

Rule 2-4 — *Requirements shall be uniquely identified.*

Each verification requirement must have a unique identifier. That identifier can then be used in cross-referencing the functional specification document, testcase implementation and functional coverage points.

Rule 2-5 — *Requirement identifiers shall never be reused within the same project.*

As the project progresses and the design and verification specifications are modified, design requirements will be added, modified or removed. Corresponding verification requirements will have to be added, modified or removed. When adding new requirements, never reuse the identifier of previously removed verification requirements to avoid confusion between the obsolete and new requirements.

Rule 2-6 — *Requirements shall refer to the design requirement or specification documents.*

The completeness of the functional verification requirements is a critical aspect of the verification process. Any verification requirement that is missing may cause a functional failure in the design to go unnoticed. Cross-referencing the functional verification requirements with the design specification will help ensure that all functional aspects of the design are included in the verification plan.

Furthermore, verification is complex enough without verifying something that is not ultimately relevant to the final design. If something is not specified, don’t verify it. Do not confuse this idea with an incomplete specification. The former is a "don’t care." The latter is a problem that must be fixed.

Recommendation 2-7 — *Requirements should be ranked.*

Not all requirements are created equal. Some are critical to the correct operation of the design, others may be worked around if they fail to operate. Yet others are optional and included for speculative functionality of the end product.

Ranking the requirements lets them be prioritized. Resources should be allocated to the most important requirements first. The decision to tape-out the design should similarly be taken when the most important functional verification requirements have been met.

Example 2-3. Ethernet IP Core Verification Requirements Ranking

```
R3.1/14/0 Packets are limited to MAXFL bytes SHOULD
R3.1/14/1 Packets can be up to 64kB          SHOULD
R3.1/14/2 Packets can be up to 1500 bytes    MUST
```

Recommendation 2-8 — *Requirements should be ordered.*

Many requirements depend on the correct operation of other requirements. The latter requirements must be verified first. Dependencies between requirements should be documented.

For example, verifying that all configuration registers can be correctly written to must be completed before verifying the different configurations.

Example 2-4. Ethernet IP Core Verification Requirements Order

```
R4.2.3/1  Frames are transmitted
R3.1/13/0 Does not append a CRC
R3.1/14/2 Packets can be up to 1500 bytes
R3.1/13/1 Appends a valid CRC
```

Recommendation 2-9 — *The requirements should be translated into a functional coverage model.*

A functional coverage model allows the automatic tracking of the progress of the verification implementation. This model also enables a coverage-driven verification strategy that can leverage automation in the verification environment to minimize the amount of code necessary to implement all of the requirements of the functional verification. More details on functional coverage modeling and coverage-driven verification can be found in Chapter 6. For example, Example 6-8 shows how the requirements shown in Example 2-5 can be translated into a functional coverage model.

Example 2-5. Ethernet IP Core Verification Requirements Coverage Model

```
R3.1/14/0 Packets are limited to MAXFL bytes
          At least one packet transmitted with length:
                  MAXFL-4
                  MAXFL-1
                  MAXFL
                  MAXFL+1
                  MAXFL+4
                  65535
          MAXFL set to
                  Default (1536)
                  1518
                  1500
          HUGEN set to
                  0
                  1
          Cross coverage of
                  frame length x MAXFL x HUGEN value
```

Recommendation 2-10 —*Implementation-specific requirements should be specified using coverage properties.*

Some functional verification requirements are dictated by the implementation chosen by the designer and are not apparent in the functional specification of the design. These functional verification requirements create corner cases that only the designer is aware of. These requirements should be specified in the RTL code itself through coverage properties.

For example, the nature of the chosen implementation may have introduced a FIFO (first-in, first-out). Even though the presence of this FIFO is not apparent in the design specification, it still must be verified. The designer should specify coverage properties to ensure that the design was verified to operate properly when the FIFO was filled and emptied.

Note that if the FIFO was never supposed to be completely filled, as assertion should be used on the *FIFO is Full* state instead of a coverage property.

Verification Environment Requirements

The primary aim of this step is to define the requirements of the verification infrastructure necessary to produce a design with a high degree of probability of being bug-free. Based on the requirements identified in the previous step, this step identifies the resources required to implement the verification requirements.

Rule 2-11 — *Design partitions to be verified independently shall be identified.*

Not all design partitions are created equal. Some implement critical functionality with little system-level controllability or observability. Others implement day-to-day transformations that are immediately observable on the output streams. The physical hierarchy of a design is architected to make the specification of the individual components more natural and to ease their integration, not to balance the relative complexity of their functional verification.

Some functional verification requirements will be easier to meet by verifying a portion of the design on its own. Greater controllability and observability can be achieved on smaller designs. But smaller partitions also increases their number, which increases the number of verification environments that must be created and increases the verification requirements of their integration.

The functional controllability and observability needs for each verification requirement must be weighed against the cost brought about by creating additional partitions.

Recommendation 2-12 —*Reusable verification components should be identified.*

Every independently verified design presents a set of interfaces that must be driven or monitored by the verification environment. A subset of those interfaces will also be presented by system-level verification of combinations of the independently-verified designs.

Interfaces that are shared across multiple designs—whether industry-standard or custom-designed—should share the same transactors to drive or monitor them. This sharing will reduce the number of unique verification components that will need to be developed to build all of the required verification environments. To that end, it will often be beneficial for the designs themselves to use common interfaces to facilitate the implementation of the functional verification task.

Different designs that share the same physical interfaces may have different verification requirements. The verification components for those interfaces must be able to meet all of those requirements to be reusable across these environments.

The opportunity for reusable verification components may reside at higher layers of abstraction. Even if physical interfaces are different, they may transport the same protocol information. The protocol layer that is common to those interfaces should be captured in a separate, reusable verification component. For example, MII (media-independent interface) and RMII (reduced media-independent interface) are two

physical interfaces for transporting Ethernet media access controller (MAC) frames. Although they have different signals, they transport the same data formats and obey the same MAC-layer protocol rules. Thus, they should share the same MAC frame descriptor and MAC-layer transactor.

Recommendation 2-13 —*Models of the design at various levels of abstraction should be identified.*

Many functional verification requirements do not need a model of the detailed implementation—such as a RTL or gate-level model—to be met. Furthermore, requirements that can be met with a model at a higher level of abstraction can do so with much greater performance. Some requirements, such as software validation, can be met only with difficulty if only an implementation-level model is available. Having a choice of models of the design at various levels of abstraction can greatly improve the efficiency the verification process.

Rule 2-14 — *The supported design configurations shall be identified.*

Designs often support multiple configurations. The verification may focus on only a subset of the configurable parameters first and expand later on. Similarly, the design may implement some configurable aspect before others with the verification process designed to follow a similar evolution.

The configuration selection mechanism should also be identified. Are only a handful of predetermined configurations going to be verified? Or, is the configuration going to be randomly selected? If the configuration is randomly selected, what are the relevant combinations of configurable parameters that should be covered? Constraints can be used to limit a random configuration to a supported configuration. Functional coverage should be used to record which configurations were verified.

Some configurable aspects require compile-time modification of the RTL code. For example, setting a top-level parameter to define the number of ports on the device or the number of bits in a physical interface is performed at compile-time. The model of the design is then elaborated to match. In these conditions, it is not possible to randomize the configuration because, by the time it is possible to invoke the `randomize()` method on a configuration descriptor, the model of the design has already been elaborated—and configured.

A two-pass randomization of the design configuration may have to be used. On the first pass, the configuration descriptor is randomized and an RTL parameter-setting configuration file is written. On the second pass, the RTL parameter-setting configuration file is loaded with the model of the design, and the same seed as in the

first pass, is reused to ensure the same configuration descriptor—used by the verification environment—is randomly generated.

Example 2-6. Supported Ethernet IP Core Configurations

```
- Variable number of TxBD (TX_BD_NUM)
    - If TX_BD_NUM == 0x00: TX disabled
    - If TX_BD_NUM == 0x80: Rx disabled
- MAXFL (PACKETLEN.15-0, MODER.14)
- Optional CRC auto-append (MODER.13)
```

Rule 2-15 — *The response-checking mechanisms shall be identified.*

The functional verification requirements describe what the design is supposed to do when operating correctly. It should also specify what kind of failures the design should not exhibit. The self-checking structure can easily determine if the right thing was performed. But it is much more difficult to determine that no wrong things were done in the process. For example, matching an observed packet with one of the expected packets is simple: If none match, the observed packet is obviously wrong. But if one matches, the question remains: Was it the packet that should have come out next?

The self-checking mechanisms can report only failures against expectations. The more obvious the symptoms of failures are, the easier it is to verify the response of the design. The self-checking mechanisms should be selected based on the anticipated failures they are designed to catch and the symptoms they present.

Some failures may be obvious at the signal level. This type of response is most efficiently verified using assertions. Some failures may be obvious as soon as an output transaction is observed. This type of response checking can be efficiently implemented using a scoreboarding mechanism. Other failures, such as performance measurements or fairness of accesses to shared resources, require statistical analysis of an output trace. This type of response checking is more efficiently implemented using an offline checking mechanism.

The available self-checking mechanisms are often dictated by the available means of predicting the expected response of the design. The existence of a reference or mathematical model may make an offline checking mechanism more cost effective than using a scoreboard.

Rule 2-16 — *Stimulus requirements shall be identified.*

It will be necessary to apply certain stimulus sequences or put the design into certain states to meet many of the functional verification requirements. Putting the design into a certain state is performed by applying a certain stimulus sequence. Thus, the problem is reduced to the ability of creating specific stimulus sequences to meet the functional verification requirements.

Traditionally, a directed sequence of stimulus was used to meet those requirements. However, the methodology presented in this book recommends the use of random generators to automatically generate stimulus to avoid writing a large number of directed testcases. Should the random stimulus fail to generate the required stimulus sequences, they are to be constrained to increase the probability that they will generate them in subsequent simulations.

The ability to constrain a random generator to create the required stimulus sequences does not happen by accident. Generators must be designed based on the stimulus sequences required by the verification requirements. They must offer the mechanisms to express constraints that will ultimately force the designed stimulus patterns to be generated as part of a longer random stimulus stream. If it remains unlikely that the required stimulus sequence will be randomly generated, then a directed stimulus sequence has to be used.

Example 2-7. Ethernet IP Core Stimulus Requirements

```
- Random configuration
  - Maximum packet length (PACKETLEN.MAXFL, MODER.HUGEN)
  - Appending of CRC (MODER.CRCEN)
  - Number of transmit buffer descriptors (TX_BD_NUM)
- Transmitted Packets
  - With and without CRC (TxBD.CRC)
  - Good & bad CRC
    - Bad CRC only if MODER.CRCEN or TxBD.CRC == 0
  - Various length (tied to maximum packet length)
```

Rule 2-17 — *Trivial tests shall be identified.*

Trivial tests are those that are run on the design first. Their objective is not to meet functional verification requirements, but to ascertain that the basic functionality of the design operates correctly before performing more exhaustive tests. Performing a write cycle followed by a read cycle, injecting a single packet or executing a series of null opcodes are examples of trivial tests.

Trivial tests need not be directed. Just as they can be used to determine the basic liveliness of the design, they can be used to determine the basic correctness of the verification environment. A trivial test may be a simple constrained-random test constrained to run for only a very short stimulus sequence. For example, a trivial test could be constrained to last for only two cycles: the first one being a write cycle, the second one a read cycle and both cycles constrained to use the same address.

The verification environment must be designed to support the creation of the trivial tests.

Example 2-8. Trivial Tests for Ethernet IP Core

```
- Rx disabled, transmit 1 packet
- Tx disabled, receive 1 packet
```

Recommendation 2-18 —*Error injection mechanisms should be defined.*

Designs may be able to sustain certain types of errors or stimulus exceptions. These exceptions must be identified as well as the mechanism for injecting them. Then, it is necessary to specify how correctness of the response to the exception will be determined.

Based on the verification requirements, it is necessary to identify the error injection mechanisms required in the verification environment. For low-level designs, it may be possible to inject errors at the same time as stimulus is being generated. For more complex systems with layered protocols, low-level errors are often impossible to accurately describe from the top of the protocol stack. Furthermore, there may be errors that have to be injected independent of the presence of high-level stimulus.

Exceptions may need to be synchronized with others stimulus—such as interrupt requests synchronized with various stages in the decode pipeline. Synchronizing an exception stream with a data stream may require using a multi-stream generator (see “Multi-Stream Generation” on page 236).

Example 2-9. Ethernet IP Core Error Injections

```
- Collisions at various symbol offsets (early, latest
early, earliest late, late)
- Collide on all transmit attempts
- Bad CRC on Rx frame
- Bad DA on Rx frame when MODER.PRO == 0
```

Recommendation 2-19 —*Data sampling interfaces for functional coverage should be identified.*

A functional coverage model should be used to track the progress toward the fulfilment of the functional verification requirements. The functional coverage model will monitor the verification environment and the design to ensure that each requirement has been met.

This monitoring requires that a *signature* is used in the design or verification environment to indicate that a particular verification requirement has been met. By observing the signature, the functional coverage model can record that the requirement corresponding to the signature's coverage point has been met.

For the coverage model to be able to observe those signatures, there must be set data sampling mechanisms. These mechanisms let the relevant data be observed by the functional coverage model. Data that is in different sampling domains or at the wrong level of abstraction will require a significant amount of processing before it can be considered suitable for the functional coverage model. Planning for a suitable data sampling interface up front will simplify the implementation of the functional coverage model.

Example 2-10. Coverage Sampling Interfaces for Ethernet IP Core

```
- DUT configuration descriptor
- Tx Frame after writing to TxBD
- TxBD configuration descriptor after Tx frame written
```

Recommendation 2-20 —*Tests to be ported across environments should be identified.*

To verify the correctness of the design integration or its integrity at different implementation stages, it may be necessary to port some tests to different environments. For example, some block-level tests may need to be ported to the system-level environment. Another example is the ability to take a simulation test and reuse it in a lab set up on the actual device. Yet another example is the ability to reproduce a problem identified in the lab in the simulation environment.

Tests that must be portable will likely have to be restricted to common features available in the different environments they will execute in. It is unlikely that these tests will be able to arbitrarily stress the capability of the design as much as a particular environment allows them to. Due to the different functional verification requirements met by the different verification environments, it is not realistic to expect to be able to port all tests from one environment to another.

Verification Implementation Plan

The primary aim of implementing the functional verification plan is to ensure that the implementation culminates in exhaustive coverage of the design and its functionality within the project time scales. The implementation is based on the requirements of the verification environments as outlined above.

This implementation plan should be started as early as possible in the project life cycle. Ideally, it should be completed before the start of the RTL-coding phase of the project and before any verification testbench code is written. This step is necessary to produce a design with a high degree of probability of being bug-free.

Recommendation 2-21 —*Functional coverage groups and coverage properties should be identified.*

The functional verification requirements should be translated into a functional coverage model to automatically track the progress of the verification project. A functional coverage model is implemented using a combination of `covergroup` or `cover property`. Which one is used depends on the nature of the available data sampling interface and the complexity of the coverage points.

Coverage properties are better at sampling signal-level data in the design based on a clock signal. But they can implement only a single coverage point. Coverage groups are better at sampling high-level data in the verification environment and can implement multiple coverage points that use the same sampling interface

Chapter 6 provides more guidelines for implementing functional coverage models.

Recommendation 2-22 —*Configuration and simulation management mechanisms should be defined.*

It must be easy—not just possible—to reproduce a simulation. It is necessary that there be a simple mechanism for ensuring that the exact model configuration used in a simulation be known. Which version of what source files, tools and libraries were used? Similarly, it must be simple to record and reissue the exact simulation command that was previously used —especially using the same random seed. Command-line options cannot be source-controlled. Therefore a script should be used to set detailed command-line options based on broad, high-level simulation options.

Recommendation 2-23 —*Constrainable dimensions in random generators should be defined.*

The random generators must be able to be constrained to generate the required stimulus sequences. Constraining the generators may involve defining sequences of data. But it also may involve coordinating multiple independent data streams onto a single physical channel or parallel channels, each stream itself made up of data sequence patterns. Constraints, state variables and synchronization events may need to be shared by multiple generator instances.

Controlability of the randomization process require the careful design of the data and transaction descriptor structures that are randomized and the generators that randomize them. The ability to constrain the generated data to create detailed stimulus scenarios tends to require more complex randomization processes. It may be more efficient to leave a few complex stimulus sequences as directed stimulus, and leave the bulk of the data generation to a simple randomization process.

Recommendation 2-24 —*Stimulus sequences unlikely to be randomly generated should be identified.*

There are some stimulus requirements that will remain unlikely to be automatically generated. Rather than complicate the random generators to create them or have to specify an overly complicated set of constraints to coerce the generators, it may be easier to specify them as directed stimulus sequences.

Directed stimulus sequences need not be for the entire duration of a simulation. They may be randomly injected as part of a random stimulus stream.

Recommendation 2-25 —*End-of-test conditions should be identified.*

When a test is considered done is an apparently simple but important question. Running for a constant amount of time or data may hide a problem located in a deeper state or by data being constantly pushed out by the forward pressure created be subsequent stimulus. Additional termination conditions could be defined: once a certain number of error messages have been reported, once a certain level of coverage has been hit, a watchdog timer has expired or the design going idle—whatever *idle* is must also be defined. The end-of-test condition could be created by only one condition or require a combination of the termination conditions.

Even when the end-of-test condition has been identified, how the simulation ends gracefully should be specified. There may be data that must be drained from the

design or statistics registers to be read. The contents of memories may have to be dumped. The testbench may have to wait until the design becomes idle.

Example 2-11. End of Test Conditions for Ethernet IP Core

```
- After N frames have been transmitted
- After M frames have been received
- If TxEN and interrupt not asserted for more than X
cycles
```

RESPONSE CHECKING

Rule 2-2 requires the enumeration of all errors that must be detected by the verification environment. These detection mechanisms require a strategy for predicting the expected response and to compare the observed response against those expectations. This section focuses on these strategies. Guidelines for the implementation of the self-checking structure can be found in section titled "Self-Checking Structures" on page 246.

It is difficult to describe a methodology for checking the response of a design because that response is unique to that design. Response checking can be described only in general terms. A broad outline of various self-checking structures can be specified. The availability in the SystemVerilog language of high-level data structures greatly facilitates the implementation of response checking. But it is not possible to describe the details of its overall implementation without targeting it to a specific design.

With traditional directed testcases, because the stimulus and functionality of the design are known, the expected response may be intellectually derived up front and hard-coded as part of the directed test. With random stimulus, although the functionality is known, the applied stimulus is not. The expected response must by computed based on the configuration and functionality of the design. The observed response is then compared against the computed response for correctness.

It is important to realize that the response-checking structure in a verification environment can only identify problems. Correctness is inferred from the failure to find inconsistencies. If the response-checking structure does not explicitly check for a particular symptom of failure, it will remain undetected. The functional verification requirements must include a definition of all possible symptoms of failure.

Recommendation 2-26 —*Response checking should be separate from stimulus.*

In directed tests, the response can be hardcoded in parallel with the stimulus. Thus, it can be implemented in a more ad-hoc or distributed fashion, in the same `program` that implements the stimulus. However, it is better to treat the response checking as an independent function.

Response checking that is hardcoded with the stimulus tends to focus on the symptoms of failure of the feature targeted by the directed test. This coding style causes functionality to be repeatedly checked in tests that focus on the same feature. But a test targeting a specific feature may happen to exercise an unrelated fault. If the response checking is concerned only with the feature being verified, then the failure will not be detected. This style of response checking may allow errors to go unnoticed if they occur in another functional aspect of the design.

By separating the checking from the stimulus, all symptoms of failures can be verified at all times.

Embedded Monitors

Response is generally understood as being observed on the external outputs of the design under verification. However, limiting response to external interfaces only may make it difficult to identify some symptoms of failure. If the verification environment does not have a sufficient degree of observability over the design, much effort may be spent trying to determine the correctness to an internal design structure because it is too far removed from the external interfaces. This problem is particularly evident in systems where internal buses or functional units may not be directly observable from the outside.

Suggestion 2-27 — *Design components can be replaced by transactors.*

Transactors need not be limited to interfacing with external interfaces. Like embedded generators described in section titled "Embedded Stimulus" on page 226, monitors can mirror or even replace an internal design unit and provide observability over that unit’s interfaces. The transaction-level interface of the embedded monitor remains externally accessible, making the mirrored or replaced unit interfaces logically external.

For example, an embedded RAM block could be replaced with a reactive transactor (slave), as illustrated in Figure 2-1. Correctness could be determined, not by dumping

or replicating the entire content of the memory, but by observing and fulfilling—potentially injecting errors—each memory access in real time.

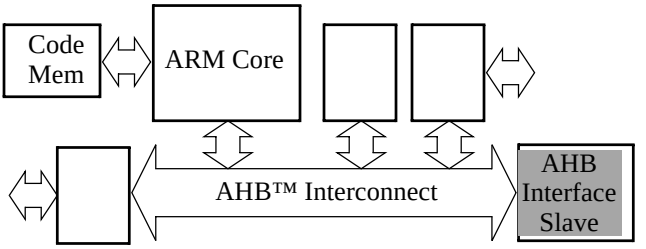


Figure 2-1. Replacing a Slave Unit with a Reactive Transactor

This approach is implementation-dependent. As recommended by Recommendation 2-32, assertions should be used to verify the response based on internal signals. However, assertions may not provide the high-level capabilities required to check the response. Assertions are also unable to provide stimulus, and thus cannot be used to replace a reactive transactor.

Assertions

The term *assertion* means a statement that is true. From a verification perspective, an assertion is a statement of the expected behavior. Any detected discrepancy in the observed behavior results in an error. Based on that definition, the entire testbench is just one big assertion: It is a statement of the expected behavior of the entire design. But in design verification—and in this book—assertion refers to a *property* expressed using a *temporal expression*.

Using assertions to detect and debug functional errors has proven to be very effective as the errors are reported near—both in space and time—the ultimate cause of the functional defect. But despite their effectiveness, assertions are limited to the types of properties that can be expressed using clocked temporal expressions. Some statements about the expected behavior of the design must still be expressed—or are easier to express—using behavioral code. Assertions and behavioral checks can be combined to work cooperatively. For example, a protocol checker can use assertions to describe the lower-level signaling protocol and use behavioral code to describe the higher-level, transaction-oriented properties.

Assertions work well for verifying local signal relationships. They can efficiently detect errors in handshaking, state transitions and physical-level protocol rules. They can also easily identify unexpected or spurious conditions. On the other hand, assertions are not well suited for detecting data transformation, computation and ordering errors. For example, assertions have difficulties verifying that all valid

packets are routed to the appropriate output ports according to their respective priorities.

The following guidelines will help identify which response-checking requirements should be implemented using assertions or behaviorally in the verification environment. More details on using assertions can be found in Chapters 3 and 7. Typical response-checking structures in verification environments are described in “Scoreboarding” on page 38, “Reference Model” on page 39 and “Offline Checking” on page 40.

Recommendation 2-28 —*Assertions should be limited to verifying physical-level assumptions and responses.*

Temporal expressions are best suited to cycle-based physical-level relationships. Although temporal expressions can be stated in terms of events representing high-level protocol events, the absence of a clock reference makes them more difficult to state correctly. Furthermore, the information may already be readily available in a transactor, making the implementation of a behavioral check often simpler.

Rule 2-29 — *A response-checking requirement that must be met on different levels of abstraction of the design shall be implemented using procedural code in the verification environment.*

Assertions are highly dependent on RTL or gate-level models. They cannot be easily ported to transaction-level models. Any response checking that must be performed at various abstraction levels of the design is better implemented in the testbench.

Recommendation 2-30 —*Response checking involving data storage, computations, transformations or ordering should be implemented in the verification environment.*

Temporal expressions are not very good at expressing large data storage requirements (such as ordering checks) and complex computations (such as a cyclic redundancy checks). Data transformation usually involves complex computations and some form of data storage. Data ordering checks involve multiple dimension queuing models. These types of checks are usually better implemented in the verification environment.

Rule 2-31 — *Responses to be checked using formal analysis shall be implemented using assertions.*

Formal tools cannot reason on arbitrary procedural code. They usually understand RTL coding style and temporal expressions. Some design structures are best verified using formal tools. Their expected response must be specified using assertions.

Recommendation 2-32 —*Response checking involving signals internal to the design should be implemented using assertions.*

Some symptoms of failures are not obvious at the boundary of the design. The failure is better detected on the internal structure implementing the desired functionality. The expressiveness of the temporal expressions usually makes such *white-box* verification easier to implement using assertions. Furthermore, being functionality internal to the design, it is unlikely that the equivalent check already exists in—or could be leveraged from—procedural code in a transactor. The response may also be interesting to verify using formal tools, as described in Chapter 7.

Recommendation 2-33 —*Assumptions made or required by the implementation on input signals should be checked using assertions.*

Often, the implementation depends on some assumed or required behavior of its input signals. Any violation of these assumptions or requirements will likely cause the implementation to misbehave. Tracing the functional failure of a design, which is observed on its outputs, to invalid assumptions or behavior on its input signals is time-consuming. These assertions capture the designer's assumptions and knowledge. They can be reused whenever the design is reused and detect errors should it be reused in a context where an assumption or requirement no longer hold. The assumptions and requirements may also be required to successfully verify the implementation using formal tools, as described in Chapter 7.

Recommendation 2-34 —*Implementation-based assertions should be specified in-line with the design by implementation engineers.*

Assertions that are implied or assumed by a particular implementation of the design are specific to that implementation. Different implementations may have different implications or assumptions. Therefore, they should be embedded with the RTL code. They can be captured only by the designer as they require specific knowledge about the implementation that only the designer has.

Rule 2-35 — *Assertions shall be based on a requirement of the design or the implementation.*

Assertions must be used to verify the intent of the design, not the language used to implement it. They must be considered similar to comments and not simply capture the obvious behavior of the language, as illustrated in the following example:

Example 2-12. Trivial and Obvious Assertion

```
always @ (posedge clk)
   i <= i + 1;
...
a: assert property (
   @(posedge clk) i == $past(i) + 1
);
```

Accuracy

The simplest comparison function compares the observed output of the design with the predicted output on a cycle-by-cycle basis. But this approach requires that the response be accurately predicted down to the cycle level, a complex task. If the design specification does not specify a particular end-to-end latency, why verify at a more accurate level of precision?

The layered verification environment (see section titled "Testbench Architecture" on page 104) allows the separation of verifying the timing from the content. The verification of the content of the design output can easily be performed with complete accuracy: Either the content of the output matches the expected content or it does not. The verification of the timing of the design output can easily sustain irrelevant variations. It may occur at different times, but as long as the output eventually comes within acceptable time boundaries, no error is reported.

The timing of physical interfaces can also be verified separately from the data being transported. Transactors can verify that the relative placement of signal transitions fall within acceptable bounds, as specified by the protocol. But they do not verify that these transitions occur at specific points in absolute time.

Ordering and sequencing are other aspects of accuracy. In some classes of designs, it may be difficult to predict the exact order in which the output transactions will be observed. Similarly, it may be difficult to determine in advance which particular transactions will be dropped to maintain some higher priority functions in the design. Rather than trying to predict the exact sequence of the output, it may be sufficient to predict the relative order of independent streams of transactions or simply assume that any transaction not observed on the output was dropped. Of course, any assumption

that could mask a functional defect should be independently confirmed through other means during the verification process.

Rule 2-36 — *Response checking shall not be more accurate than necessary.*

If it is not specified, don't check for it. Suggestion 2-41 and Suggestion 2-42 describe types of behavior that may be checked with varying degrees of accuracy.

Recommendation 2-37 —*Response checking should be transaction accurate.*

The response should be verified based on the correctness of the transaction data. The timing of transactions should only be verified with respect to the occurrence of other transactions i.e., sequencing, ordering and maximum latency.

Recommendation 2-38 —*Only interfaces should be checked for timing accuracy.*

Transactors monitoring an interface should check that the timing of the signals on that interface is internally consistent and timing accurate. The relative position of signal transitions should fall within acceptable bounds but not verified against an absolute time reference.

Interfaces should not be checked cycle by cycle to allow for nonfunctional variations. For example, whether a read cycle introduces zero or several wait states is not functionally relevant—unless the function being verified is the performance of the interface.

Recommendation 2-39 —*The relative timing of different interfaces should not be verified.*

The relative timing of signal transitions on different interfaces should not be verified, unless some specified relationship exists between the interfaces.

Recommendation 2-40 —*Cycle-level accuracy should be checked only when the specification is stated at the cycle level.*

If the functional verification requirement includes a cycle-level check of the response or throughput of a design, then these requirements trump all previously stated recommendations in this section. If it is specified, it must be verified.

Suggestion 2-41 — *It may not be necessary to predict the exact transaction execution order.*

This suggestion is a special case of Rule 2-36. It may be sufficient to verify that some relative order is maintained. For example, check that independent streams multiplexed onto a single output stream are in order, but do not attempt to predict the exact inter-stream ordering. Another example would be out-of-order processor instructions: As long as instructions are executed in order of data dependencies, the exact execution order may not need to be predicted.

Suggestion 2-42 — *It may not be necessary to predict exactly which transaction will be dropped.*

This suggestion is a special case of Rule 2-36. In some applications—e.g., network routers, transactions can be dropped as part of normal operations of the designs. Is it important to predict which transaction will be dropped? Or that, if transactions are observed to have been dropped, that the minimum number of transactions were dropped and for the right reasons, regardless of which ones were dropped? For example, would it be important to predict which packets were dropped to meet quality-of-service requirements? Or would it be sufficient to check that those packets that were dropped belong to the lowest quality-of-service class?

Instead of predicting which transaction will be dropped, it may be sufficient to identify that transactions were dropped and that it occurred if and only if a valid condition was present. Assertions can be used to detect the occurrence and duration of drop conditions, isolating the verification environment from implementation details.

Scoreboarding

A scoreboard is used to dynamically predict the response of the design. As illustrated in Figure 2-2, the stimulus applied to the design is concurrently provided to a transfer function. The transfer function performs all transformation operations on the stimulus to produce the form of the final response then inserts it in a data structure. Observed response from the stimulus is forwarded to the compare function to verify that it is an expected response.

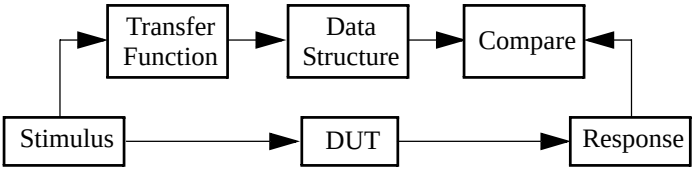


Figure 2-2. Scoreboarding

The transfer function is a transaction-level reference model that usually operates in zero time. It may also be implemented using a reference or golden model. The data

structure stores the expected response until it can be compared against the observed output. The compare function looks up the expected response in the data structure to identify if the observed response matches expectations. The data structure and compare function handle any acceptable discrepancy between the observed response and the expected output, such as ordering or latency.

The transfer function and data structure are usually configurable to match the configuration of the DUT: Different configurations may yield different responses. Transfer functions may be implemented in C. The *Direct Programming Interface* may be used to integrate them in the SystemVerilog environment. A directed test may implement its expected response using a test-specific transfer function that models only the necessary subset of the functionality that is exercised.

The term “scoreboard” is not well-defined in the industry. It sometimes refers to the storage data structure only, sometimes it includes the transfer function as well, and sometimes it includes the comparison function. In this book, the term *scoreboard* is used to refer to the entire dynamic response-checking structure.

Scoreboarding works well for verifying the end-to-end response of a design and the integrity of the output data. It can efficiently detect errors in data computations, transformation and ordering. It can also easily identify missing or spurious data. On the other hand, it is not well suited for detecting errors whose symptoms of failures are not obvious at the granularity of a single response. For example, scoreboarding has difficulty verifying the fairness of internal resource allocations and quality-of-service arbitrations. It may also be difficult to use a scoreboard to measure overall performance of the design under verification.

Reference Model

A reference model, like a scoreboard, is used to dynamically predict the response of the design. As illustrated in Figure 2-3, the stimulus applied to the design is concurrently provided the reference model. The output of the reference model is compared against the observed response.

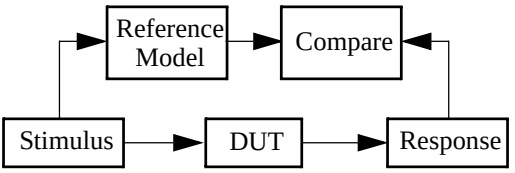


Figure 2-3. Reference Model

Reference models have the same capabilities and challenges as scoreboards. Unlike a scoreboard, the comparison function works directly from the output of the reference model. The reference model must thus produce output in the same order as the design itself. However, there is no need to produce the output with the same latency or cycle accuracy: The comparison function can handle latency and cycle discrepancies between the expected and observed response. A reference model need not be pin-accurate with the design. A reference model can be at the transaction level, with a high-level transaction interface: The comparison of the observed response with the response of the reference model is performed at the transaction level, not at the cycle-by-cycle level.

Using reference models depends heavily on their availability. If they are available, they should be used. If they are not available, scoreboarding techniques will be more efficient to implement. More often than transfer functions, reference models are implemented in C. The *Direct Programming Interface* may be used to integrate them in the SystemVerilog environment.

Offline Checking

Offline checking is used to predict the response of the design before or after the simulation of the design. As illustrated in Figure 2-4, in a pre-simulation prediction, the offline checker produces a description of the expected response, which is dynamically verified against the observed response during simulation. The compare function can dynamically compare the predicted response to the observed response or a utility can perform the comparison post-simulation. As illustrated in Figure 2-5, in a post-simulation prediction, the recorded stimulus and response of the design is compared against the predicted result by the offline response checker. In both cases, the response can be checked at varying degrees of details and accuracy, from cycle-by-cycle to transaction-level with reordering.

Using pre-simulation response prediction with dynamic response checking lets a simulation report any discrepancy while the design is in or near the state where the error occurs. It also avoids needlessly running long simulations when a fatal error occurs early in the run. Pre-simulation checking cannot generate stimulus based on the dynamic state of the design—such as the insertion of wait states—and may not exercise the design under all possible conditions.

Offline checking works well for verifying the end-to-end response of a design and the integrity of the output data based on executable system-level specifications or mathematical models. It can efficiently detect errors in data computations, transformation and ordering. Offline checking can also easily identify missing or spurious data. Post-simulation offline checking is also well suited for detecting errors

whose symptoms of failures are not obvious at the granularity of a single response. For example, it can verify the fairness of internal resource allocations and quality-of-service arbitrations by performing statistical analysis over the entire recorded response.

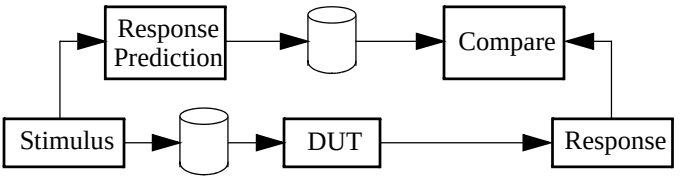


Figure 2-4. Pre-Simulation Offline Checking

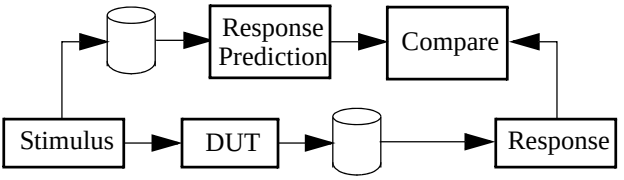


Figure 2-5. Post-Simulation Offline Checking

Offline checking need not be implemented separately from the runtime simulation environment. The invocation of external programs necessary to generate the input, predict the response and compare it with the observed response can be done by the simulator at the start or the end of the simulations. Although offline checking is usually used with a reference model, it can be used with scoreboarding techniques implemented as a separate offline program.

SUMMARY

This chapter described the necessary steps required to plan a verification project. First, the requirements that must be met by the verification projects are defined. These requirements create specifications for the stimulus, response-checking and functional coverage aspects of the verification environment.

Next, various strategies for computing or specifying the expected response of a design under verification were presented. The different strategies have different advantages and limitations when comparing the observed response against expected results. A particular strategy may have to be used to identify certain classes of failures, which may not be as easily identifiable in another approach.

Assertions are best for verifying implementation-specific and physical-level relationships; whereas testbenches are best for verifying transaction-level responses. Unlike testbenches, assertions are not limited to the primary DUT outputs to check its response. The complete response of a design will be verified using a combination of assertions and one or more verification environments.

CHAPTER 3 ASSERTIONS

Concurrent and immediate assertions are an integral part of SystemVerilog. They provide an effective way to improve observability and localization of design errors by placing functional checks at critical points inside a design and on external interfaces. When an assertion fails during simulation, the cause of the reported error can be more easily identified than by observing simulation traces. Concurrent `assert property` statements can also be proven or falsified using formal tools, thus further increasing confidence in the quality of the design. Finally, *coverage properties* provides information on how well the design has been functionally exercised during simulation.

This chapter first introduces SystemVerilog concurrent assertions and their possible uses. This introduction will be useful to readers new to the notion of assertions. Next, this chapter examines the role of assertion on internal signals and interfaces of a design. This will be useful to designers, because it requires detailed knowledge of the design. Assertions on external interfaces useful to both verification and design engineers, are also examined.

This chapter presents several guidelines for writing efficient assertions and coverage properties. Guidelines and techniques for packaging assertions into reusable checkers are also presented. Using checkers simplifies the process of adopting assertions. Checkers can be divided into two broad categories: basic checkers and assertion-based verification IP. Basic checkers will be of interest to anyone who plans to use checkers or develop new ones. The section on assertion-based verification IP is mainly directed toward developers of such IP. This chapter ends with a discussion on how to verify assertion-based checkers.

SPECIFYING ASSERTIONS

Assertions are observers that monitor signals in the DUT for correct behavior. They could be implemented as Verilog modules or VHDL entity-architecture pairs, like the checkers in the Open Verification library (OVL). Writing assertions for complex sequential behaviors using Verilog or VHDL is difficult. Worse, there is the risk that an assertion will be implemented in the same manner as the function it is to verify and thus contain the same errors. These are the main reasons specialized languages for writing assertions have been developed.

In SystemVerilog, unlike the procedural code of the DUT or testbench, the assertions are expressed using a declarative language. The declarative language was influenced by a number of similar predecessor languages—Intel's *ForSpec* language, Motorola's CBV and Synopsys' OpenVera Assertions (OVA)—and the Accellera *Property Specification Language* (PSL) in an effort to realign the two assertion languages. SystemVerilog's assertions bring a number of innovations that are particularly useful in dynamic verification, such as:

- the well-defined *sampling* of all variables in the simulation cycle that avoids races between the design, assertions and the verification environment,
- *local variables* that are dynamically allocated during property evaluation, providing much increased modelling power,
- the *action blocks* which can be attached to `assert` and `cover` statements. It executes whenever the statement succeeds or fails, providing a flexible way to communicate with the testbench or even other assertions,
- the possibility of *triggering the execution* of non-blocking tasks at any point within assertion sequences, and
- *recursive properties* which open new ways to formulate properties.

Assertions can influence many parts of the verification process, as shown in Figure 3-1. Whether written using the SystemVerilog assertion language to describe a particular behavior or by using a checker from the SystemVerilog Checker library, the application of assertions remains the same. Assertions formalize the unambiguous specification of protocols used in the design and can verify that the DUT satisfies the specified behavior using simulation, emulation or formal means.

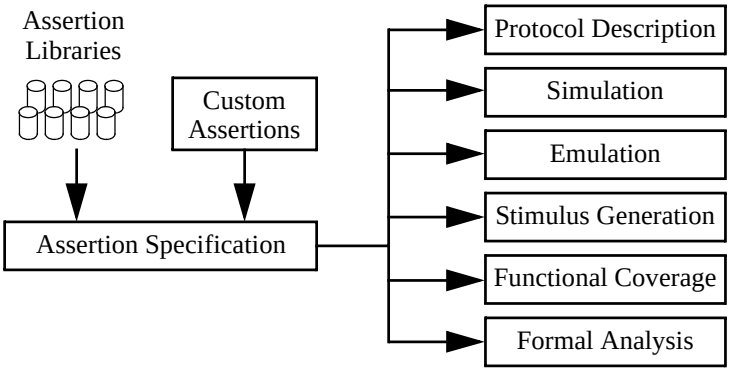


Figure 3-1. Assertions in a Verification Process

In formal tools, some properties stated using the *assert property* statement are used as assertions to be proven on the DUT. These proofs are usually subject to other properties that describe the assumed behavior of the environment using *assume property* statements. In simulation, asserted and assumed properties are continuously verified to ensure that the design or the testbench never violate them. In some tools, the assumptions on the environment can be used as sequential constraints on the DUT inputs in constrained-random simulation. Functional coverage specified using the *cover property* statement can confirm that specific corner cases have been exercised in simulation or, using formal tools, it can be determined if the corner cases can be reached and then generate a test to activate them.

Assertions can be parameterized for reuse in different contexts. Assertions can be packaged in *modules* or *interfaces* to create a reusable checker library. Such a library usually deals with properties suitable for checking generic behaviors in any design. Sets of assertions describing standard protocols such as PCI, PCI Express, AMBA AHB™ Protocol, Utopia, etc. can be packaged to create reusable assertion-based verification IP units. The verification IP includes a complete set of checks and coverage points for a particular standard protocol. Implementing such assertion-based verification IP is described in the section titled "Assertion-Based Verification IP" on page 86.

Assertions in a DUT can be applied along one of two categories: on internal signals of the design, including inter-block interfaces and on external interfaces of the design, either standard or custom

Applying assertions to internal signals requires detailed knowledge of the internal structure of the design. Therefore, the DUT designers are best placed to insert such assertions and coverage points. Essentially, the assertions and coverage points represent a form of active comments within the design. The next subsection covers this application of assertions.

Applying assertions to external interfaces treats the DUT as a black box. It is concerned with the correct function of the design, regardless of its implementation. These assertions are usually written by verification engineers as part of the testbench development. If the external interface is an industry or company standard, assertion-based protocol verification IP may be used. The issues related to using assertions on external interfaces are discussed later in this chapter.

Assertion Language Primer

This chapter requires some basic understanding of the SystemVerilog assertion language operators. The reader should consult the *SystemVerilog Language Reference Manual* and other books for more details on this subject.

Table 3-1. Summary of Sequence Operators

`##M` `##[M:N]`[a]	*Concatenation*. For example, "`a ##[1:4] b`" states that `b` must follow `a` within one to four clock ticks.
`[*M]` `[*M:N]`[a]	*Repetition*. For example, "`s[*1:3]`" states that `s` is repeated between one to three times, at each clock tick. This is equivalent to writing `s or (s ##1 s) or (s ##1 s ##1 s)`.
`b[->M]` `b[->M:N]`[a]	*Goto repetition*. For example, "`expr[->1]`" is equivalent to writing "`(!expr)[*0:$] ##1 expr`," which states: match on the first occurrence of `expr` being true, including the case where `expr` is true at the current clock tick.
`b[=M]` `b[=M:N]`[a]	*Non-consecutive repetition*. For example, "`expr[=1]`" is equivalent to writing "`(!expr)[*0:$] ##1 expr ##1 (!expr)[*0:$]`". The difference is the trailing repetition of "`!expr`". It is useful in situations where some number of occurrences of the Boolean expression are allowed to be true anywhere within some other temporal sequence, e.g. when using the `within` operator.

a. where N >= M >= 0, and N can be $ to specify *open ended*

Table 3-1. Cont.

b throughout s	*Throughout*. For example, "*req throughout (grant[->1])*" states that *req* must remain asserted until *grant* becomes asserted.
s1 within s2	*Within*. The sequence expression will match if *s1* occurred at least once anywhere between the start and end of *s2*. For example, "*(expr[=3]) within (a[->1])*" states that *expr* must be true for three (possibly non-consecutive) clock ticks while awaiting the occurrence of *a* becoming true.
s1 intersect s2	*Intersect*. *s1* starts at the same time as *s2*; The intersection will match if *s1*, starting at the same time as *s2*, matches at the same time as *s2* matches. For example, "*1'b1[*3:5] intersect (a ##0 b[->1] ##0 c[->1])*, states that *a* being true in the current cycle must be followed by *b* being true at the same time or later, which in turn must be followed by *c* being true at the same time as *b* or later, and the entire sequence from *a* to *c* must not last more than three to five cycles.
s1 and s2	*And*. With *s1* starting at the same time as *s2*, the sequence expression matches with the later of *s1* and *s2* matching. For example, "*(a ##[1:3] b) and (c ##2 d)*" states that this sequence will match at cycle 3 after *a* and *c* being true at the same time followed by *b* being true one or two cycles later and *d* being true two cycles after *c*. It will also match at cycle 4 after *a* and *c* being true at the same time followed by *b* being true three cycles later and *d* being true two cycles after *c*.
s1 or s2	*Or*. The sequence expression matches when the first of s1 or s2 matches. For example, "*(a ##[1:3] b) or (c ##2 d)*" states that this sequence will match at cycle 2 when *b* is true one cycle after *a* being true. It will also match at cycle 3 when *a* is true followed by *b* true two cycles later, or *c* is true followed by *d* true two cycles later, etc.

Sequence operators construct *sequence* expressions from Boolean expressions. Boolean expressions are sequence expressions of length one. Sequences are used in properties for use in assertions and covers. Sequence operators are summarized in Table 3-1.

Property operators construct properties out of sequence expressions. A sequence can be promoted to a property. Property operators are summarized in Table 3-2.

Once instantiated (or inlined) in an `assert property`, `cover property` or `assume property` statement, every property or part thereof must have a clock associated with it. The clock can be specified directly in the property or in the `property` statement. It can also be inferred from the surrounding context of a clocked `always` block or a `default clocking` block specification.

Table 3-2. Summary of Property Operators

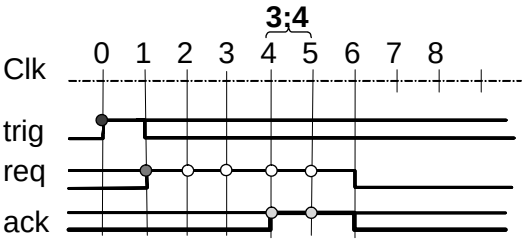
`s |-> p` `s |=> p`	*Implications*. Whenever the antecedent sequence `s` matches, the consequent property `p` must be true starting at the clock tick where `s` matched, or to the nearest future clock tick after `s` matched, respectively. For example, the property "`$rose(trig) ##1 req \|=> req[*3:5] \|-> ack`" will be *vacuously* true when the antecedent does not match because "`$rose(trig)`" is false, or "`$rose(trig)`" is true but it is not followed by `req` true one cycle later, or "`$rose(trig)`" is true followed one cycle later by `req` true but in the subsequent two cycles `req` becomes false. It will be *non-vacuously* true when, for every match of the antecedent sequence, the property is true: "`$rose(trig)`" is true followed one cycle later by `req` true and one cycle later `req` is true for three to five cycles, and then `ack` is asserted for each of these cycles that `req` is asserted beyond the first two cycles. An example of a satisfying trace (waveform) is shown in Figure 3-2.  **Figure 3-2.** A Satisfying Trace for Example Property
`not p`	*Not*. For example, "`req |-> not (##[1:3]ack)`" states that, when `req` is true, `ack` must not be true within three cycles.

Table 3-2. Cont.

`p1 and p2`
`p1 or p2`
`if (b) p1 else p2`
	Property and, or, if-else. `b` is a boolean expression and `p1` and `p2` are properties. The operators combine the logical truth of the component properties as implied by the name of the logic operator.

`disable iff (b)`	*Disabling*. If the immediate value of `b` is true—it is not sampled by any clock—it preempts any evaluation attempt in progress and declares it as a vacuous success.

By default, a property will start a new evaluation attempt at every clock tick of its associated clock when it is instantiated or inlined in a `property` statements or when a sequence `s` is used with one of the pseudo-method `s.ended`, `s.triggered` or `s.matched`. If the `property` statement is placed in an `initial` block, it will evaluate only once, starting on the first clock tick.

Evaluation is carried out over the values of the variables sampled in the *preponed* scheduling region and the immediate values of local variables in the property. The preponed region is located at the beginning of a time step, before the *active* region. Therefore, assertions use the stable value design variables have reached just before the next clock edge.

Depending on the structure of the property, every evaluation attempt creates one or more evaluation threads. Multiple threads are created when the property involves choices induced by intervals in concatenations or repetitions, or by the use of the `or` operator, and when more than one thread continues evaluation because the boolean expressions evaluate true.

A sequence *matches* on the observed trace when it successfully reaches (one of) its end point(s). Note that for a given evaluation attempt, a sequence may have more than one match due to multiple threads reaching their end points.

An `assert property` or `assume property` statement succeeds when its evaluation arrives at the conclusion *true*. It fails otherwise. For example, if the property is a sequence, then for a given evaluation attempt the property succeeds on the first match of the sequence. It fails for a given evaluation attempt if the sequence has no match. In properties involving implications, for the property to succeed, the consequent must evaluate true for every matching thread of the antecedent sequence.

A `cover property` over a property *matches* if the property succeeds, a `cover property` over a sequence *matches* whenever the sequence matches.

ASSERTIONS ON INTERNAL DUT SIGNALS

Assertions and coverage points on internal signals are inserted by the *designer* during the detailed implementation of a block. This section contains some guidelines on typical areas in the design that should be instrumented with assertions. This list is not exhaustive as it is not possible to enumerate all design structures. However, a basic rule is that any design code that embodies non-trivial behavior over time should be instrumented with assertions.

Recommendation 3-1 — *Internal assertions should be used instead of comments.*

An assertion, with its failure message, serves as a form of documentation—an "active" comment—that describes in both formal and natural languages the expected behavior of the signals involved in the assertion. If that behavior is not respected, the assertion fails and the failure reports the violation. Of course, not all comments can be turned into an assertion. But where practical, internal assertions will provide increased observability and immediate identification of structural design errors. Something a comment cannot do. Trying to identify the ultimate cause of an error by observing signal waveforms is much more tedious. Assertions shorten the debug time.

Recommendation 3-2 — *Internal corner cases should be identified using coverage properties or coverage group.*

To ensure the completeness of the verification suite, designers should identify all corner cases in the implementation using a `cover property` statement or a `covergroup` instead of specifying a testcase for the verification team to write. Functional coverage points provide automatic feedback about the yet unverified corner cases of the design. There is no need to rely on the verification plan documentation process AND the continued correct implementation of a testcase to verify implementation-specific corner cases. This guideline is consistent with Recommendation 2-10.

Recommendation 3-3 — *Assertions on internal signals should be placed with the design code.*

Internal assertions express requirements or functionality that may not be apparent in the design specification. They may be the consequence of the chosen implementation architecture or an arbitrary communication protocol between two design blocks. The assertion statements should be placed in the RTL code where they apply because they are intimately tied to that code and must be maintained in concert. It is possible to

inline the assertion or coverage point close to the point in the design for easier understanding, as shown in Example 3-1.

Example 3-1. Inlined Concurrent Assertion

```
module moduleA(input bit clk, output ...);
   ... // design code
   check_progress : assert property (
   @(posedge clk) disable iff (reset)
   (valid_in && !stall) ##1 (!stall[->3]) |-> ##1 f3 );
   ... // design code
endmodule : moduleA
```

For consistency with Recommendation 3-1, inlined assertions should be located near the RTL code implementing the behavior they verify. In doing so, it is important that the implementation does not taint the assertion implementation. If the same error is duplicated in the assertion, it will not detect the corresponding invalid behavior. Alternatively, the assertions could be located together at the top or bottom of the file.

Recommendation 3-4 — *Assertions inside `always` blocks should be used with caution.*

If the signals involved in the assertion are assigned in a synchronous `always` block, then placing the assertion directly inside the procedural code lets the sampling clock and the enabling condition be extracted from the `always` block code. For example, the assertions in Example 3-2 and Example 3-3 are functionally equivalent.

Example 3-2. Extracting Clock and Enabling Condition From `always` Block

```
always @(posedge clk)
  if (reset) st_in <= 0;
  else
    if (!st_in && ready_in) begin : set_st_in
      st_in <= 1;
      ...
      st_hold: assert property(
        ##1 (st_in [*1:$]) ##0 accepted_in)
           else
        $display("st_in did not hold till accepted_in");
    end : set_st_in
    else
      if (st_in && accepted_in) st_in <= 0;
```

Example 3-3. Equivalent Concurrent Assertion

```
st_hold: assert property( @(posedge clk)
  (!reset && (!st_in && ready_in))
    |->
     ##1 (st_in [*1:$]) ##0 accepted_in)
  else
$display("st_in did not hold till accepted_in");
```

Although syntactically convenient, there is some danger in inferring the enabling condition from the RTL code. If the condition is in fact incorrect, the assertion may not detect the error. For instance, consider the case when the condition over *`!st_in && ready_in`* is incorrect. In that case the assertion will be triggered, like the RTL code, by the incorrect condition and will succeed if *`st_in`* and *`accepted_in`* are still computed correctly. If it were written independently, the condition in the assertion would be derived from the required behavior and through this difference trigger the assertion at a different time than the computation in the always block. That does not guarantee detection, but it may increase its likelihood.

Recommendation 3-5 — *Inlined assertions should be embedded in `‘ifdef` blocks.*

If a particular tool does not support inlined assertions, they can be automatically eliminated by defining or undefining the corresponding pre-processor symbol. The standard SYNTHESIS symbols should not be used because some synthesis tools may support assertions.

Example 3-4. Controlling the Introduction of Inlined Assertions

```
...
‘ifndef NO_INLINED_ASSERTION
   st_hold: assert property( @(posedge clk)
     (!reset && (!st_in && ready_in))
         |-> ##1 (st_in [*1:$]) ##0 accepted_in)
   else $display("st_in did not hold till accepted_in");
‘endif
...
```

Rule 3-6 — *Assertions for normal DUT operations shall be disabled when reset is in progress.*

Unless the assertion targets the behavior during or right after reset, the behavior embodied in the assertion may be violated. Therefore, the assertion should be disabled. There are two possible ways to disable an assertion. One way is to use the

disable iff clause in the assertion, as shown in Example 3-5. This kind of disabling can also be used with formal tools.

Example 3-5. Reset in `disable iff`

```
check_progress : assert property (
   @(posedge clk) disable iff (reset)
      (valid_in && !stall) ##1 (!stall[->3])
       |-> ##1 f3 );
```

The other way is to turn off all assertions that should not run during reset using the system task *$assertoff* before entering reset, and then once reset is over, enable them using the system task *$asserton*. However, this works only in simulation and not with formal tools.

Recommendation 3-7 — *The checkers from the VMM checker library should be used wherever possible.*

Using predefined checkers has a number of advantages as compared to writing custom assertions: the amount of coding is reduced, custom checkers may not be easy to write and debug by novice users and the standard checkers have been thoroughly verified.

As long as there is a predefined checker matching the required behavior, they provide a quick way to add assertions to the design. The standard VMM Checker Library currently has 51 checkers. These include 31 checkers that are equivalent—including having the same name—to the well-known Accellera OVL. Most of the remaining 20 checkers cover more complex behaviors such as arbiters, memories and FIFOs.

The VMM checkers can verify behaviors from very simple boolean invariant properties like *assert_always* to medium complexity time-based properties like *assert_window* and handshaking properties like *assert_handshake* or *assert_valid_id* to complex hardware blocks such as FIFOs, stacks, memories and arbiters. For more detail, please refer to Appendix B.

The desired behavior may not be entirely covered by a single checker in the library. It is often possible to decompose the behavior into a collection of properties that can be implemented using standard checkers. Together, they imply the overall behavior.

Note that novice users can learn how to write assertions and coverage points by examining the existing VMM checkers and tailor them to the specific needs of their project. Since the VMM checkers try to be very general, the derived specialized versions may be more efficient.

Rule 3-8 — *Assertion-based checkers shall be encapsulated using an* `interface` *construct.*

The `interface` construct should be used because it provides the most flexibility: it can be instantiated in a `module`, an `interface` or a `program`.

Rule 3-9 — *Every FSM shall have assertions that verify the state encoding and transitions.*

The checks shall verify that invalid states are not reached, that the reset state is correct and is always reached upon reset, that only valid transition sequences are taken, that outputs are correct for all states and transitions and that assumptions on inputs to the FSM are respected by its environment.

It is important, however, that the RTL code not be simply duplicated in the assertions. Any error in the RTL code would be replicated in the assertions. The FSM assertions must be specified based on the FSM requirements.

Many VMM standard checkers are suitable for verifying FSM behavior, such as:

```
assert_bits, assert_next_state, assert_value,
assert_cycle_sequence, assert_code_distance,
assert_next, assert_transition, assert_one_hot,
assert_one_cold, assert_zero_one_hot
```

The state and transition properties should also be covered to ensure that all states and transitions have been exercised.

Rule 3-10 — *Every internal block interface shall have assertions that verify the assumed interface protocol.*

Unverified assumptions used in the implementation of an interface are often a source of errors if the neighboring block violates these assumptions. Therefore, each interface should have assertions on the complete protocol.

Numerous standard VMM checkers can be used to build the appropriate assertions for an interface protocol. For example:

```
assert_change, assert_unchange, assert_hold_value,
assert_req_ack_unique, assert_valid_id,
assert_reg_loaded, assert_req_requires, assert_window,
assert_even_parity, assert_odd_parity, assert_driven,
assert_no_contention, assert_frame, assert_handshake,
assert_time, assert_win_change, assert_win_unchange.
```

Recommendation 3-11 —*Interface-related assertions or checkers should be specified in the* `interface` *declarations.*

If the interface signals and behavior are specified using the `interface` construct, the assertions should be placed inside that construct, thus assuring that, wherever that `interface` is used, the protocol is verified.

If it is an interface external to the block or design, the architecture of the checker should follow the guidelines for reusable assertion-based checkers in the section titled "Architecture of Assertion-Based IP" on page 90.

Example 3-6. Assertions in an `interface` Declaration

```
interface itf_in(input clk, reset);
  wire [7:0]  data_in;
  reg  [1:0]  id_in;
  wire        ready_in;
  wire        accepted_in;
 `ifndef NO_INLINED_ASSERTION
  a0: assert property(
    @(posedge clk) disable iff (reset)
      $rose(ready_in) ||
     (ready_in && $past(accepted_in))
        |-> ##[1:10] accepted_in )
    else
$display(“accepted_in not within 5 to 10 clock cycles”);
...
endinterface : itf_in
```

Rule 3-12 — *Every FIFO, stack or memory shall have assertions on its proper use.*

An incorrect use of a FIFO or a stack—overflow, underflow, data corruption—and invalid memory accesses—overwriting without reading, reading before writing, etc.—are common kinds of errors. Any design implementing these functions shall have assertions verifying their usage by other functions. The following checkers for these structures can be found in the standard VMM Checker Library:

```
assert_fifo, assert_dual_clk_fifo,
assert_multiport_fifo, assert_stack, assert_memory_sync,
assert_memory_async, assert_fifo_index.
```

Rule 3-13 — *Assertions shall be used to verify that arbitration for access to resources follows the appropriate rules.*

Errors in arbitration may create unfair access to a shared resource or lead to starvation of some requestors. The symptoms of these failures can be very difficult to detect at

the boundary of the design. Assertions that verify rules of the specific arbitration algorithm can be very effective in detecting such problems.

The `assert_arbiter` checker in the standard VMM Checker Library is designed to verify different arbitration algorithms: priority with or without fairness, FIFO or least-recently-used (LRU). However, there are many variants and additional arbitration rules, making it is impossible to create a completely general arbitration checker. The user can use the checker for verifying the basic functionality and enhance it with DUT specific assertions.

Rule 3-14 — *There shall be no assertion on the periodicity of the system clock itself.*

The SystemVerilog assertion language targets synchronous systems where all signals are updated synchronously with a clock. An assertion verifying the clock signal itself would have to run on simulation time as its sampling clock, which is impossible in the current formulation of the assertion language. Properties that monitor the clock signal must be implemented using procedural code.

Rule 3-15 — *There shall be no concurrent assertion to monitor combinatorial signal glitches or asynchronous timing.*

Concurrent assertions target synchronous systems where all signals are updated and sampled synchronously with a clock. Properties that monitor combinational glitches or asynchronous timing must be implemented using procedural code and immediate assertions.

Rule 3-16 — *There shall be no assertion that verifies the correctness of the SystemVerilog language or known-to-be-good components.*

Assertions should capture intent that is not obvious in the code itself. They should not verify that the tool correctly interprets the semantics of the SystemVerilog language. Such assertions are considered trivial and guaranteed to always be met in a correct implementation of the language.

Example 3-7. Trivial Assertion

```
assign     d0 = data_in + data_in;

useless_assert : assert property(
    @(posedge clk) disable iff (reset)
                  d0 == data_in + data_in );
```

Recommendation 3-17 —*Assertions should verify that arithmetic operations do not overflow and/or the target registers do not change value by more than some +/- delta.*

In many cases, the automatic truncation of overflow or underflow value is a source of errors. Similarly, large changes in a register value may be a symptom of functional errors. The following VMM standard checkers may help formulating those checks:

```
assert_no_overflow, assert_no_underflow, assert_delta,
assert_range, assert_increment, assert_decrement.
```

Rule 3-18 — *Decoding and selection logic shall have assertions to verify mutual exclusion.*

Violation of mutual exclusion may lead to bus contention and other interference between blocks targeted by the selection/decode logic. The following VMM checkers may be useful to verify such logic:

```
assert_bits, assert_code_distance, assert_one_hot,
assert_one_cold, assert_mutex, assert_no_contention,
assert_value, assert_zero_one_hot, assert_proposition,
assert_never, assert_next.
```

Rule 3-19 — *Whenever a signal is to hold for some time or until some condition occurs, such behavior shall be verified using assertions.*

For example, implementations of specification requirements such as "*once asserted,* `s` *must remain asserted for three clock cycles*" and "*once asserted,* `load` *must remain asserted until* `eop` *is asserted*" should be verified using assertions.

The standard VMM Checker Library contains a number of checkers that can be used in this context:

```
assert_always, assert_never, assert_unchange,
assert_change, assert_hold_value, assert_reg_loaded,
assert_req_resp, assert_time, assert_window,
assert_width, assert_win_change, assert_win_unchange.
```

Rule 3-20 — *Any time-bounded well-defined relationship between signals shall be checked using assertions.*

For example, implementations of specification requirements such as "When `data_valid_in` is asserted, `data_valid_out` will be eventually asserted, within a certain amount of time," or "signal `a` asserted implies that condition `c` holds within one to three clock cycles." Assertions should be used to verify the stated cause-effect relationships.

Many VMM standard checkers can be used to verify such relationships:

```
assert_always, assert_never, assert_req_requires,
assert_always_on_edge, assert_change, assert_frame,
assert_implication, assert unchange, assert_win_change,
assert_win_unchange, assert_next.
```

Rule 3-21 — *During a reset, conditions on control signals and shared buses shall be verified using assertions.*

Such checks shall verify that a reset operation lasts at least the required minimum number of cycles, that bus drivers are tri-stated or, equivalently, that *enable* signals on bus drivers are de-asserted. As per Rule 3-15, this rule assumes *resets* span at least one clock cycle and can thus be sampled by the assertion. In addition to the VMM standard checkers mentioned in Rule 3-20, the following are likely to be useful:

```
assert_no_contention, assert_driven.
```

It is also possible to verify that, when exiting asynchronous reset, signals have a certain value just before the deactivating transition on the reset. Example 3-8 assumes that *reset* is asynchronous and active high and that the signal *data_out* must be tri-stated before exiting the reset condition. Note that this assertion cannot be used with formal tools that do not accept case equivalence relations (`===`, `!==`) which are non-synthesizable.

Example 3-8. Using Reset to Verify Signal Values

```
check_z : assert property(
    @(negedge reset) (data_out === 8'bzzzzzzzz) );
```

The assertion in Example 3-8 verifies the intended behavior because *data_out* is sampled in the preponed region of the time step in which `negedge reset` occurs. Thus its value is the one just before the deactivating transition on *reset*.

Suggestion 3-22 — *A `category` attribute on `assert` and `cover property` statements may be used for tool-specific control of assertions in simulation.*

A *category* attribute on `assert` and `cover property` statements allows tools that recognize this attribute to provide additional control of assertions that is beyond the simple hierarchical controls available through the standard system tasks `$assertoff`, `$asserton` and `$assertkill`.

For example, all assertions that carry some specific category value could be started and stopped. Failures or coverage reports could be sorted based on category values. If

the category values are mapped to the test plan sections, these controls can select assertions and covers according to the test plan hierarchy rather than the design hierarchy.

Tools that do not support this attribute will ignore it. In Example 3-9, the *category* attribute is assigned a parameter of the same name.

Example 3-9. Using *category* Attribute

```
(* category = category *) assert_stack_hi_water_chk:
    assert property( @( posedge sva_checker_clk)
      disable iff( !not_resetting)
        not( $rose(sva_v_stack_ptr > hi_water_mark)))
    else sva_checker_error("");
```

Rule 3-23 — *A specific failure message of an* `assert property` *statement shall be produced through a VMM message service interface in the action block of the assertion.*

This rule applies if the assertions are running with a VMM compliant testbench. In that case an instance of the VMM message service interface class must be created in the *module* or *interface* containing the assertion. Whereas the assertion may be synthesizable, the message service interface is not. A preprocessor symbol can be used to select the appropriate configuration for the tool as shown in Example 3-10. The *vmm_log* class and its usage are as described in "Message Service" on page 134.

Example 3-10. Using VMM Message Service Interface for Reporting

```
`ifndef SYNTHESIS
  vmm_log log = new("pipeline checks", $psprintf(%m));
....
  check_z : assert property(
     @(negedge reset) data_out === 8'bzzzzzzzz )
  else
     `vmm_error(log, "data_out not Hi-Z when in reset");
‘endif
```

ASSERTIONS ON EXTERNAL INTERFACES

Assertions on external interfaces are used by verification engineers. The assertions can be created by them or can already be provided by designers on block interfaces that become exposed as external system interfaces. Assertions on external interfaces are derived from the functional specification of the interface signals and protocols and

coverage statements are related to the verification plan, rather than the internal structural features of the blocks. The DUT is generally viewed as a black box. However, specifying certain assertions may be require access the enable signals of shared bus drivers, as explained in the rules governing reusable assertion-based checkers in section titled "Reusable Assertion-Based Checkers" on page 77.

Many of the guidelines regarding assertions on internal signals and interfaces apply to assertions on external interfaces. The guidelines related to the use of checkers, the form of properties in `cover property` statements, and failure reporting are all applicable.

Rule 3-24 — *External interface assertions shall be attached to the DUT module using the* `bind` *statement or become part of the* `interface`*.*

Because the DUT is treated as a black box, internal signals are not visible and no code can be inserted in the design by verification engineers. The `bind` statement provides a non-intrusive means for attaching checkers without any design code modification. Also, the separation of the verification code simplifies maintenance. If the interface signals are specified using an `interface`, then the assertions can simply be placed inside the interface and thus become part of its specification.

Rule 3-25 — *Assertions shall be divided into two categories: assertions on local interface protocols and assertions on signals from two or more interfaces.*

The first category is concerned with local behavior of the interface protocol. This type of assertion makes sure that communication follows the specified rules. Often, these protocol assertions are suitable for verification by formal tools.

The second category is concerned with end-to-end properties of the DUT, relating some conditions on one of its interfaces with conditions on some other interface. Such assertions may cover a very small portion of the behavior and be amenable to formal proofs, but more often the portion of the design determining the behavior to be verified by the assertion is too big for the formal tool. The assertions covering the global behavior may be suitable only for simulation or for searching for bugs using formal techniques. Furthermore, end-to-end properties may be easier to check in the testbench.

Recommendation 3-26 —*Custom assertions verifying local protocols should follow rules for constructing reusable assertion-based checkers.*

Even if the interface protocol is not standard and may not be reused on another design, the reusability rules assure that the checker can be used both for block-level verification with formal tools—where some assertions become assumptions on the environment—and at block or system-level simulations—where all interface assertions verify the communication on interconnections among blocks.

Recommendation 3-27 —*Assertions involving end-to-end behavior of the DUT may reuse definitions and variables from the local interface protocol checkers.*

Assertions that verify some global end-to-end behavior of the DUT may deal with more coarse information units like bits assembled into bytes, or bytes into words, etc. Sequences and checker auxiliary variables that assemble required information may already exist in the local interface checkers and should be reused.

Suggestion 3-28 — *Checks over global behavior of the DUT at the transaction level may be better implemented using scoreboarding techniques.*

At the transaction level, the behavior is often expressed in terms of information exchanges and transformations using (potentially large) blocks of data or transaction descriptors. The events that determine the validity of the generated or received descriptors are often on a different time scale than the system clock of the design. Also, their decoding, checking and routing may involve complex algorithms. These may not be easily expressible in assertions and can be difficult to prove by formal tools due to their complexity.

In general, if the check involves extensive data structures or algorithms and spans a large portion of the design, checking by using testbench monitors is more appropriate.

Recommendation 3-29 —*Physical-level assertions can be used to notify testbench monitors.*

Interface protocol checkers can be used to validate the protocol and if an error is detected, these checkers can notify the testbench monitors verifying the application-level data. For example, if an assertion fails, a flag can be set in the action block of the assertion. That flag can be watched by the testbench monitor. The monitor can take

appropriate action—such as suspending monitoring activities for the remainder of the transaction—if the flag becomes set.

Example 3-11. Notifying a Monitor from an Assertion

```
interface mii_sva_checker (
  reset_n, TX_CLK, ...,
  TX_FrameError, ...);
...
  task TX_SetFrameError(input mii_sva_tx_error_type i);
    TX_FrameError = i | TX_FrameError;
  endtask : TX_SetFrameError
...
  mii_TX_3: assert property(mii_3_p)
     else begin
       sva_checker_error(“ ... “);
       TX_SetFrameError(MII_TX_NO_CRS_W_COL);
     end
...
endinterface : mii_sva_checker
```

In Example 3-11, the failure of assertion *mii_TX_2* is reported using the *sva_checker_error* task (which may use the *vmm_log* facility for reporting). A bit corresponding to that error in the *TX_FrameError* signal is then set. The testbench can then react to a change of value of the flag to disable the monitor that is currently extracting the frame the data. The testbench is responsible for resetting the flag.

In some cases, it may be useful to clear the flag only once the system has recovered, e.g. after the reception of a new packet. Alternately, the simulation could be stopped if the failure is fatal.

The flags can also be used to disable other assertions and covers, by using a reduction *or* over the flag bits inside the `disable iff` clause of the appropriate assertions. Example 3-12 illustrates that organization.

Example 3-12. Using Flags to Disable Other Assertions

```
interface mii_sva_checker (
  reset_n, TX_CLK, ...,
  TX_FrameError, ...);
...
  task TX_SetFrameError(input mii_sva_tx_error_type i);
    TX_FrameError = i | TX_FrameError;
  endtask : TX_SetFrameError
...
  property mii_TX_3_p;
```

```
      disable iff (resetting || (| TX_FrameError))
        $rose(TX_EN) |-> !CRS;
    endproperty : mii_TX_3_p
    mii_TX_3: assert property(mii_3_p)
       else begin
         sva_checker_error(“ ... “);
         TX_SetFrameError(MII_TX_NO_CRS_W_COL);
       end
// similarly in other assertions and covers
// related to the TX function
...
endinterface : mii_sva_checker
```

Notice that as long as at least one bit gets set in the `TX_FrameError` vector, the assertions related to the TX function are disabled. Hence, the first assertion that fails disables all the other ones. Once the testbench clears the *`TX_FrameError`* flags, the assertions are re-enabled.

ASSERTION CODING GUIDELINES

This section describes some basic guidelines for writing assertions to be verified using simulation. Note that Chapter 7 contains additional guidelines that should be followed when the assertions are to be used with formal tools or emulation systems. Additional guidelines are required because the assertions may have to be compiled to synthesizable RTL code.

Rule 3-30 — *Open-ended interval `##[n:$]` shall be constrained by other operators.*

Failures of assertions involving a time shift with an open-ended interval to model eventuality cannot be detected using simulation. At best, a simulator will indicate that the evaluation attempt of the assertion did not finish before the end of the simulation run.

Example 3-13. Eventuality Property

```
a14: assert property( @(clk) a |-> ##[1:$] b );
```

The assertion in Example 3-13 cannot fail in simulation and, unless a *b* is sampled true after *a* is sampled true, the evaluation attempt will remain active until the end of the simulation run. Nevertheless, an indication that *b* is missing can be deduced from the fact that the evaluation attempt of the assertion has not completed by the end of simulation. If *a* becomes true often without *b* becoming true at all, the simulation

may have to maintain a large number of active evaluation threads, which can affect simulation performance.

Similarly, the following negated property cannot succeed:

Example 3-14. Negated Property

```
a15: assert property ( @(clk) not (a ##[1:$] b) );
```

The property in Example 3-14 states that whenever *a* is true, it must never be followed by a *b* being true. Unless it fails (i.e., *b* is true after *a*), the evaluation attempt will remain active until the end of simulation. If many such attempts are created, severe simulation performance problems could arise[1].

Open-ended intervals are useful when constrained by other operators.

Example 3-15. Constraining by Intersection to Limit Length

```
a_length: assert property
     ( @(posedge clk)
       y |-> 1'b1 [*20:30]
               intersect
             (a ##[1:$] b ##[1:$] c)
     ) else ...;
```

This property in Example 3-15 states that after *y*, *a* must be followed by *b,* then must be followed by *c*, and the total extent of the sequence *a* ... *b* ... *c* is at least 20 and at most 30 clock ticks.

Similarly, using an open-ended interval together with the *within* operator also limits the extent of the eventuality as shown in the example below.

Example 3-16. Bounding Length Using *within*

```
a_within: assert property
     ( @(posedge clk)
       y |-> (a ##[1:$] b ##[1:$] c)
                 within
               x[->2]
     ) else ...;
```

1. Most formal tools can detect a failure of the above negated property, but may not do the same in the case of the positive assertion shown in Example 3-13, unless the tool can verify unbounded liveness.

The property in Example 3-16 states that whenever *y* occurs, then within the first occurrence of *x* after *y* the sequence *a* ... *b* ... *c* must be detected at least once. Note that if *x* remains *false* until the end of simulation, the evaluation attempt will never fail and will remain active until the end of the run. Therefore, it may be preferable to verify (possibly by a separate assertion) that *x* will effectively occur after a *y* within some specific number of clock cycles. If only one occurrence of the sequence on the left-hand side of *within* should appear, the property could be written as follows:

```
a_within: assert property
     ( @(posedge clk)
       y |-> (a[->1] ##1 b[->1] ##1 c[=1])
                intersect
              x[->2]
     ) else ...;
```

In this case, there must be no additional occurrence of *c* after the sequences is detected within *x[->2]*.

Rule 3-31 — *Open-ended ## intervals shall not be used in antecedent sequences of an implication without other constraints.*

Consider the property *s0 ##[M:$] s1 |=> s2*.

If the property is used in an assertion that is not inside an *initial* block (i.e., it is always activated) then at every clock tick a new attempt is started. In each evaluation attempt for the property to hold, whenever *s0 ##[M:$] s1* matches, *s2* must also match. However, due to the open-ended interval, it will match on every occurrence of *s1* that follows *s0*, and it will search for such an occurrence of *s1* until the end of simulation. If *s0* occurs many times during the simulation, all of these attempts and threads will remain active and, unless *s2* does not match after a match of the antecedent thus causing a failure, the evaluation attempts will remain undecided until the end of simulation. With the ever increasing number of evaluation threads, the simulation load may increase quite considerably.

It is often the case that *s1* in the above example is a boolean expression *b*. The problem with this form is that an evaluation attempt of *b* is started any time *s0* matches, but that attempt will remain active, searching for *b* to be true until the end of simulation and the attempt will never terminate unless *s2* fails after *b*. Yet, the probable intent was to detect only the first match of *b2* after *s0*. This detection can be accomplished in the following way:

Example 3-17. Using the go-to Operator in Place of Eventuality

```
a_go_to: assert property( @(posedge clk)
       (s0 ##1 b[->1]) |=> s2
  ) else ...;
```

Notice how, in Example 3-17, *s1* was modified using the *go-to [->]* operator to match only on the first occurrence of *b* strictly after *s0*. If *s1* is a temporal sequence, then the *go-to* operator cannot be used and the solution is to use the *first_match* operator, as in Example 3-18. However, this operator is less efficient.

Example 3-18. Using *first_match* Operator

```
a_first_match: assert property ( @(posedge clk)
      s0 ##1 first_match( ##[1:$] s1 ) |=> s2
  ) else ...;
```

Rule 3-32 — *The $past system function shall not be used over a large number of clock cycles.*

Large number of clock cycles may reduce verification performance because the *$past* operator resembles a shift register with as many stages as the delay argument. What is a large number depends on the specific tool, but it is a good practice to avoid using *$past* with more than a few clock cycles.

Consider reformulating the same property without the use of a deep look into the past using that operator. Often this reformulation can be achieved by identifying a condition that marks the instant when the value is valid. Use that condition to store the value in a variable for use later. Note, however, that overlapping transactions (e.g., a pipeline) require using local variables or a FIFO object.

Recommendation 3-33 —*A large time window should be delimited using variables.*

Large time intervals in the *##[M:N]* and *[*M:N]* operators may require long compilation times or use a lot of runtime resources. If the bound is large (> 20), consider rewriting the property in some other way. For example, express such properties with the help of variables.

Suppose that after *start*, the signal *stop* should be asserted within 150 to 256 clock ticks. A direct but potentially inefficient way to code this property would be to use *##[150:256]*:

Example 3-19. Coding with a Large Time Shift Interval

```
a1: assert property(@(clk) start |-> ##[150:256] stop);
```

If the protocol is such that after a `start` is issued, a `stop` must be received before the next start (i.e., there are no overlapping `start-stop` sequences), then the property should be coded using a variable. Additional assertions should be specified to verify the non-overlapping assumption of the `start-stop` sequences. Notice that a missing `stop` is also detected because the assertion will also fail as soon as the `timer` reaches the upper limit of 256.

Example 3-20. Non-Overlapping Transactions with a Large Time Interval

```
logic [10:0] timer = 0;
always @(posedge clk) begin : nooverlap
  if (reset || stop)   timer <= 0;
  else if (start)      timer <= 1;
  else if (timer != 0 && timer <= 255)
    timer <= timer + 1;

  timing: assert property(
    disable iff (reset)
    $rose(stop) || $rose(timer == 256)
      |->  (timer > 150) && (timer <= 255) )
  ) else ...;
end : nooverlap
```

In the case of overlapped transactions, the long *##* interval can be avoided by using local variables. For example, if `start` is asserted then `stop` must be asserted within the next *n >= 0* cycles. This can be verified using the property shown in Example 3-21. Note that *n* may be a register in the design.

Example 3-21. Overlapping Transactions with a Large Time Interval

```
property overlap_p;
   logic [10:0] timer;
   @(posedge clk)
     (start, timer = n) |->
       ((timer > 0, timer=timer-1)[*0:$] ##1 stop);

endproperty : overlap_p
timer_overlap:  assert property (overlap_p) else ...;
```

Suggestion 3-34 — *It may be preferable to code assignments to auxiliary variables using ternary `?:` expressions.*

For some assertions, it is important to detect the presence of `X` or `Z` states on signals. Using the `?:` operator will propagate any unknown on the predicate condition to the final result. If an `if-else` statement is used, as in Example 3-20, the *else* result would be used and the unknown condition would not be detected by the assertions

using that result. Example 3-20 could be re-coded using *?:* as shown in Example 3-22.

Example 3-22. Using Ternary *?:* Expressions to Propagate Unknowns

```
logic [10:0] timer3 = 0;
always @(posedge clk) begin : nooverlap_ternary
  timer3 <=
    (reset || stop) ? 0 :
      start ? 1 :
       (timer3 != 0 && timer3 <= 255) ? timer3 + 1 :
         timer3;
```

Recommendation 3-35 —*Reduction and word-level operators should be used to simplify boolean expressions in sequences.*

It is often necessary to test for specific values of bits in a bit vector. Instead of comparing each individual bit, it is often possible to use reduction operators and word-level operators to simplify the boolean expression and thus improve verification tool performance.

For example, to test that all bits in a vector *W* are set to 1, use *(&W)* instead of *W == '1.* To test that there is an odd number of bits in *W*, use *(^W)*. To verify that there is at least one bit set in the same position in words *W1* and *W2*, use *|(W1 & W2)*.

Recommendation 3-36 —*The use of boolean expressions should be given preference over edge expressions.*

Edge expressions add overhead during verification, and in many cases, edge detection is not needed. For example, to check that a pulse on signal *x* is only one clock cycle wide can be written as:

```
$rose(x) |-> ##1 $fell(x);
```

However, a simpler equivalent check can be stated as:

```
x |-> ##1 !x;
```

Rule 3-37 — *An implication shall not be used in a negated property.*

The form *s1 |-> s2* is used in a positive assertion, while the form *s1 |-> not(s2)* is used in a negative assertion that forbids *s2* from occurring. For example, suppose it is necessary to verify that when *a* occurs, then it is not followed

by *b* in the next clock cycle. If *b* is boolean, this property could be coded as shown in Example 3-23

Example 3-23. Property with Negated Boolean in the Consequent

```
a1: assert property(@(posedge clk) a |-> ##1 !b
) else ...;
```

However, if *b* is a sequence, this property could also be expressed using the violating sequence and a negation as shown in Example 3-24.

Example 3-24. Complemented Property Containing an Implication

```
a2: assert property(@(posedge clk) not( a |-> ##1 b )
) else ...;
```

The problem is that whenever *a* is false, then *a |-> #1 b* succeeds, and hence, the negation (*not*) will fail. This is clearly not the intended result. It is required that if *a* happens and it is followed by *b* then the assertion should fail which is exactly how the property should be written:

Example 3-25. Moving Negation to the Consequent

```
a3: assert property(@(posedge clk) a |-> not( ##1 b ))
else ...;
```

Here, *a3* will report a failure only when the *a* occurs and is followed by *b*. An additional advantage of this formulation is that vacuity of the assertion can also be detected, i.e., the situation when *a* never occurs.

Recommendation 3-38 —*Variables should be used to store expected results for data checking.*

Static variables can be used to keep or compute expected values that will be compared later with monitored signals provided that multiple transactions do not overlap.

Example 3-26. Checking Expected Value in a Non-Overlapping Transaction

```
logic [7:0] v_data_in = 0;
always @(posedge clk) begin : no_overlap
  v_data_in <=
    port_in.accepted_in && all_empty ? port_in.data_in :
                                       v_data_in;
  data_check: assert property (
    disable iff (reset)
```

```
      all_empty && port_in.accepted_in |->
         ##4 (port_out.data_out == (v_data_in << 1))
    ) else $display ("...");
end : no_overlap
```

If multiple transactions overlap and may carry different data values, the mechanism used in Example 3-26 is not sufficient because the same variable is used by all overlapping transactions. Usually the transaction is also identified by some tag (or id) so that the request and the subsequent acknowledgment can be matched, as shown in Example 3-27.

Example 3-27. Checking Expected Value in Overlapping Transactions

```
property p;
  logic [7:0] v_data_in;
  logic [1:0] v_id_in;
  @(posedge clk) disable iff (reset)
    (port_in.accepted_in, v_data_in = port_in.data_in,
                          v_id_in = port_in.id_in) ##1
    (port_out.accepted_out &&
    (port_out.id_out==v_id_in))[->1]
        |->
      (port_out.data_out ==  (v_data_in<<1));
endproperty : p
overlap_data_check: assert property(p)
   else ...;
```

Rule 3-39 — *Assertions shall be disabled upon reset condition.*

When an assertion evaluation is triggered, the property expressed may not hold if, during the consequent evaluation, a reset occurs. Use the *`disable iff`* clause if a reset during the evaluation of the assertion should cancel the evaluation.

Example 3-28. Using `disable iff`

```
check_progress : assert property(
     @(posedge clk) disable iff (reset)
     (valid_in && !stall) ##1 (!stall[->3]) |-> ##1 f3
) else ...;
```

If *`reset`* becomes true while evaluating the body of the property, then the evaluation is terminated without a failure.

An alternative that may not work with formal tools is to stop the assertions before entering reset by using the *`$assertoff`* system task and re-enable them after reset using *`$asserton`*.

Suggestion 3-40 — *The operand of* `disable iff` *may be a formal argument of the property.*

The boolean expression of `disable iff` may be a formal argument of a reusable property. However, since only one top-level `disable iff` clause can appear in a `property` statement, a property that contains a `disable iff` cannot be instantiated in another property that already contains that clause.

Example 3-29. Parameterized `disable iff` Operand

```
property p(disable_bool, ... other formals... );
  disable iff(disable_bool) ... ;
endproperty: p
```

Recommendation 3-41 —*Group assertions by clock source in an* `always` *block.*

When assertions are grouped together in a file (rather than inlined in place of application), it may be more convenient to infer their clock from an `always` block. In this way, it does not have to be repeated in every assertion and the clock can be easily changed. Using a `default clocking` block is also an option, but then the same clock will apply to all assertions unless explicitly overridden.

Example 3-30. Using an `always` Block to Infer a Clock

```
logic [7:0] v_data_in = 0;
always @(posedge clk) begin : no_overlap
  v_data_in <=
    port_in.accepted_in && all_empty ? port_in.data_in :
                                       v_data_in;
  data_check: assert property (
    disable iff (reset)
    all_empty && port_in.accepted_in |->
       ##4 (port_out.data_out == (v_data_in << 1))
  ) else ... ;
end : no_overlap
```

Recommendation 3-42 —*Sequence, property and block definitions should be delimited using labels.*

To improve the readability of the assertion code (and any SystemVerilog code in general), it is preferred to label both the beginning and the end of the block using the same label.

Example 3-31. Labeling of Blocks

```
property p;
   ...
endproperty: p

sequence s;
   ...
endsequence: s

always @(clk) begin: a
   ...
   for (...) begin: f
      ...
      if (...) begin: i_true
         ...
      end: i_true
   end: f
end: a
```

Coverage Properties

Assertions are designed to specify expected behavior. Structural coverage metrics can also be gathered on *assert property* statements described in the preceding sections. Although knowing which assertions have been exercised is necessary, it is not sufficient for measuring how much the DUT has been exercised. The same language used to specified expected behavior can also be used to specify required behavior. The *cover property* statement is designed to specify functional coverage points.

Functional coverage can be subdivided into two basic classes—data/stimulus coverage and protocol/activity coverage. The former is concerned with the coverage of computed data and is best measured by using the *covergroup* statement. The latter is concerned with the coverage of the sequencing or control aspects of the DUT and is best expressed using *cover property* statements. More details on functional coverage can be found in Chapter 6.

Because of its sequential and physical-level nature, protocol coverage is more easily described and gathered by specifying temporal sequences. Each describe some characteristic protocol element that must be exercised during tests. Also, coverage properties can serve as goals for automatic test sequence generation using formal tools. However, there are certain aspects in protocol coverage that are also best implemented using *covergroup* constructs or a combination of a *sequence* and *covergroup*. These are particularly useful when covering the observed latencies between the occurrences of two conditions, packet/data lengths, etc.

Recommendation 3-43 —*Significant events should be covered using coverage properties.*

`cover property` statements on local interfaces or those on structures like FIFOs, stacks, arbiters, etc. can detect the occurrence of signal value sequences that identify significant events related to the behavior of the protocol or object. The occurrence of the significant events should be included in the functional coverage model to ensure that they have all been exercised.

Often, the occurrence of significant events is an opportunity for sampling additional data for more complex functional coverage models. Sequence definitions that detect such significant events can also be used as sampling events in `covergroups`.

Rule 3-44 — *Only sequences shall be used in* `cover property` *statements.*

If properties containing implications are used, they succeed when their antecedent fails. This success is *vacuous* and is not indicative of the success of the entire assertion. Only the non-vacuous successes—when the antecedent matches the observed behavior—signal the occurrence of the significant event associated with the property. To avoid confusion about what is vacuous, the best practice is to use only sequences in coverage properties: Sequences either match or not on the observed trace.

Example 3-32. Checking Assertion

```
a0: assert property (
  @(posedge clk) disable iff (reset)
  $rose(ready_in) || (ready_in && $past(accepted_in))
        |-> ##[1:10] accepted_in
) else $display("...");
```

To cover the specification in Example 3-32, it is preferable to express it as shown in Example 3-33.

Example 3-33. Coverage Property

```
c1: cover property (
  @(posedge clk) (!reset) throughout
  $rose(ready_in) || (ready_in && $past(accepted_in))
      ##[1:$] accepted_in );
```

If the coverage tool implements vacuous success filtering, then this rule may not be as important. However, note that in complex properties involving nested implications, the vacuity detection may not be accurate; hence, it is preferable to express the expected sequences to be covered (i.e., not as a property containing an implication).

Similar considerations apply to properties involving the *if* operator without an *else* branch. Notice also the open-ended interval when awaiting *accepted_in*. The cover is written to match on any occurrence of the handshake, its purpose is not to verify the latency.

Rule 3-45 — *Cover properties for normal DUT operations shall be disabled when reset is in progress.*

Like assertions, there are two ways to disable coverage. The same system tasks can be used as in Rule 3-6. In the other cases, it is preferable to use the *throughout* operator to disable coverage instead of *disable iff*.

Example 3-34. Using *throughout* to Disable Cover Property

```
c1: cover property (
  @(posedge clk) (!reset) throughout
  $rose(ready_in) ... );
```

Whenever *reset* becomes true, the boolean expression on the left-hand side of the *throughout* operation becomes false, thereby forcing no match of the sequence. Were *disable iff* used instead, a reset would cause a vacuous match that could confuse the reader of the coverage report unless vacuous matches are eliminated by the tool. Using *throughout* to disable a cover will not detect asynchronous reset pulses that occur between two clock ticks.

Recommendation 3-46 —*coverage properties should be used to enumerate compliance test sequences.*

The specifications of standard protocols such as the AMBA Protocol Family, PCI, etc. often contain compliance sequences that describe the types of transactions and sequences of transactions that must be supported by the DUT to be considered compliant. Using coverage properties to specify such compliance sequences can provide useful information about the progress of the interface controller verification and its level of compliance. Note that compliance statements involving data transformation properties may be better implemented using testbench monitors.

Recommendation 3-47 —*Inlined* cover property *statements and* sequence *declarations should be used to specify corner cases implied by the design.*

Corner cases are often implied by the chosen architecture of a design. They are not obvious from the design specification. These corner cases can be added to the

functional verification requirements by specifying them using inlined *cover property* statements.

Recommendation 3-48 —*Coverage measurement should be turned on in regression tests for all* `assert` *and* `cover` *statements each time the design or the testbench has been modified.*

Every time the design has changed significantly or the testbench is modified, the previous coverage data is not valid anymore and has to be reconstituted. However, if only new tests are added, coverage should be turned on only for these new tests and the results then merged with the existing coverage data.

Suggestion 3-49 — *The action statement associated with a* `cover property` *statement may be used to trigger an event.*

The event may be used by the testbench to detect that the property was covered. The event can then trigger further coverage in a *covergroup*, or a different phase in the testbench.

Example 3-35. Notifier in Failure Action Statement

```
event p5_covered;

always @p5_covered $display(“Cover c5 matched”);

property p5;
  @(posedge clk) !reset throughout
  ($rose(cnt == 1) ##[1:$] (cnt == 10));
endproperty : p5

c5: cover property(p5)
  begin
    -> p5_covered;
  end
```

Suggestion 3-50 — *The action statement associated with a* `cover property` *statement may force sampling in a* `covergroup` *instance using the* `sample()` *method.*

Rather than tie a *covergroup* to an event triggered by a coverage property, it may be simpler to trigger the coverage sampling by directly calling the *sample()* method.

Example 3-36. Covergroup Sampling from Action Statement of Cover Property

```
covergroup cg;
  coverpoint pre_cnt {
    bins which_cnt[] = {[0:15]}; }
endgroup : cg;
cg covergroup_instance = new();

// There shall be as many matches of c6 as the sum of
// all bin counts in the covergroup_instance

c6: cover property(p6) covergroup_instance.sample();
```

Suggestion 3-51 — *Sequences may be used as the sampling event in a* `covergroup`.

When a `sequence` matches the observed trace, it is possible to trigger sampling in a `covergroup`. This match becomes very useful for covering the values of local variables that were collected during the evaluation of the sequence. The following example illustrates a possible approach to covering an MII packet length:

In Example 3-37, `cnt` is a local variable that counts the number of nibbles in the frame, and `sample_TX_Length` is a task that converts the count to octets, stores it in a static variable `TX_FrameLength` and forces sampling by the `covergroup`. The `always` block outputs a message whenever the sequence matches. It is also necessary that the sequence be instantiated for it to run.

Example 3-37. Coverage of Packet Length Using a `covergroup` and a `sequence`

```
int TX_FrameLength = 0;

covergroup length_cg;
  coverpoint TX_FrameLength;
  option.per_instance = 1;
endgroup : length_cg

length_cg mii_TX_frame_length_cg = new ();

task store_cnt(input int x);
  TX_FrameLength = x;
  mii_TX_frame_length_cg.sample();
endtask : sample_TX_Length

sequence frame_length_s;
  int cnt;
  @(posedge TX_CLK) 1'b1 ##1
  ( (not_resetting && !COL) throughout
    ($rose(TX_EN), cnt=1) ##1
```

```
    (TX_EN, cnt++)[*0:$]) )
  ##1
  (!TX_EN, store_cnt(cnt));
endsequence : frame_length_s

always @(frame_length_s)
       $display("Frame length sampled");
```

Rule 3-52 — *`Cover property` statements shall not be used to implement coverage of a large set of data values.*

It is possible to implement coverage of data values using `cover property` statements embedded in one or more loops in a `generate` statement. However, although very easy to code, the resulting large number of generated c`over property` statements can considerably diminish compilation and simulation performance. The following example illustrates this case:

Example 3-38. Inefficient Use of `cover property` for Data Coverage

```
genvar i;
generate
  for (i=0; i<256; i++) begin : many_cov
    cnt_cov: cover property (
      @(posedge clk) !reset && cnt[4] &&
                     (cnt[7:0] == i[7:0]) );
  end : many_cov
endgenerate
```

In Example 3-38, the `generate` loop, once unrolled, will construct 256 *cover propertie*s that must be evaluated on every clock tick. Yet, only one will ever match at each clock tick. This kind of coverage is best implemented using a `covergroup`:

```
covergroup cg @(posedge clk)
  coverpoint cnt[7:0] iff (cnt[4] && !reset)
    { bins cnt_cov[] = {[0:255]}; }
endgroup
cg cg_inst = new();
```

REUSABLE ASSERTION-BASED CHECKERS

Reusable checkers can verify some relatively simple properties typically found in any design. They are usually included in a library of checkers. The VMM checker library is an example of simple reusable checkers. This section specifies guidelines for writing simple reusable checkers.

Some checkers can also be used as abstract models of the environment. They can be used as constraints for random test generators or assumptions for formal verification engines. Guidelines are required to write reusable checkers that can be used as abstract models as well as checkers.

Assertion IP can verify complete behaviors of specific standard protocols and behaviors, such as PCI, Utopia, SPI, AMBA Protocol Family, etc. Due to their more complex nature, additional guidelines are required for writing assertion IP. These guidelines are specified in "Assertion-Based Verification IP" on page 86.

Simple Checkers

Simple checkers are intended to verify behaviors that are typical to many designs. Thus, the checkers must be reusable in various contexts, either instantiated directly in a DUT or bound using `bind` statements. It is not expected that these checkers will appear in a testbench program; instead, these checkers may be partly associated with a `module` or an `interface`.

Each checker may contain auxiliary state variables, `cover` statements and/or `covergroups`, and of course one or more `assert` statements. Both assertions and coverage can be globally controlled on an individual checker or a whole design sub-hierarchy. Reporting of any errors or assertion failures may also be done using the message service (See "Message Service" on page 134).

Rule 3-53 — *Checkers shall be packaged using an* `interface.`

`interfaces` can be instantiated in either a `module`, an `interface` or a `program`.

Example 3-39. Checker Packaging

```
(* sva_checker *) interface assert_stack(
   clk, reset_n, push, push_data, pop, pop_data);
```

Rule 3-54 — *The inclusion of* `assert property` *statements in the checker shall be controlled by the macro* `ASSERT_ON.`

This inclusion is for compatibility with the Accellera OVL. If this macro is defined, then the assertions in the checker are compiled; otherwise, they are excluded.

Rule 3-55 — *The inclusion of* `cover property` *statements in the checker shall be controlled by the macro* `COVER_ON.`

This inclusion mechanism is similar to the `ASSERT_ON` control. If this macro is defined, then all coverage statements (`cover property` statements and `covergroup` instances) in the checker are compiled; otherwise, they are excluded.

Using the above two macros, the user can use the checker for coverage, assertions or both.

Rule 3-56 — *A checker shall have a* `category` *parameter to allow category-based control of its operation.*

There are system tasks that allow controlling checkers, not only by their position in the hierarchy, but also by their category. The `category` parameter is used as an attribute of the `assert property`, `cover property` statements and `covergroup` instances[2].

Rule 3-57 — *A global reset signal shall be optionally specified by the macro* `ASSERT_GLOBAL_RESET.`

The reset (or enable) signal can be passed through a port, but also by defining the macro ASSERT_GLOBAL_RESET. This macro can refer to some global signal or expression which is then used as the reset condition.

Example 3-40. Reset Selection

```
`ifdef ASSERT_GLOBAL_RESET
  assign not_resetting = (`ASSERT_GLOBAL_RESET != 1'b0);
`else
  assign not_resetting = ( reset_n != 0);
`endif
```

In Example 3-40, if `ASSERT_GLOBAL_RESET` is defined, then this condition—when equal to 0—becomes the reset condition. Otherwise, the reset is taken to be the port `reset_n` of the checker. For example, if the macro is defined as:

```
‘define ASSERT_GLOBAL_RESET top.reset
```

then the checker would be reset anytime `top.reset == 0`.

2. Note that the assertion attribute `category` is implemented in the Synopsys VCS simulator, but may not be supported by other tools.

Rule 3-58 — *Non-synthesizable code shall be controlled by the standard macro* `SYNTHESIS`.

Several constructs useful for simulation are not supported by synthesis. See IEEE 1364.1 *Standard for Verilog Register Transfer Level Synthesis* for a specification of the synthesizable subset.

Example 3-41. Use of `SYNTHESIS` Macro

```
`ifndef SYNTHESIS
   // non-synthesizable code here
‘endif
```

Rule 3-59 — *A checker shall create an instance of the message service interface class.*

This message service interface shall be used to issue all messages from the checker instance when the checker is used with a VMM compliant testbench.

Example 3-42. Instance of the Message Service Interface Class

```
vmm_log log = new(“assert_name”, $psprintf(“%m”));
```

The `vmm_log` based reporting is part of the messaging tasks used in the standard checkers and contained in the `sva_std_tasks.h` file.

Rule 3-60 — *The initial identification of the checker instance shall be controlled by the macro* `ASSERT_INIT_MSG`.

It is often required that each checker instance reports its existence. This initial identification must be controlled by a macro as illustrated on the following example:

Example 3-43. Issuing an Initial Checker Identification Message

```
`ifdef ASSERT_INIT_MSG
  initial sva_std_init_msg();
`endif
```

The `vmm_log` class instance and the `category` parameter are global and visible to all reporting task calls.

Rule 3-61 — *Non-synthesizable assertions shall be under the control of the macro* `SVA_CHECKER_FORMAL.`

Assertions that check for the presence of `'x` or `'z` using case equality (= = =) are not synthesizable and may not be used with formal tools. These assertions must be conditionally included or excluded

Example 3-44. Non-Synthesizable Assertion

```
`ifdef SVA_CHECKER_FORMAL
`else
   (* category = category *)assert_no_contention_no_xs :
   assert property(
     @(posedge sva_checker_clk)
     disable iff (!not_resetting)
             (^bus1 !== 1'bx) || (en_vector == 0))
   else sva_checker_error(“”);
`endif // SVA_CHECKER_FORMAL
```

Rule 3-62 — *Assertion failures shall be reported in the* `else` *part of the assertion* `action` *block using the message service.*

This rule is a consequence of Rule 4-43 on page 134. The task `sva_std_error()` shall include the assertion category in a standard way in the error message.

Example 3-45. Assertion Failure Reporting

```
task sva_checker_error;
  input bit [60*8-1:0] err_msg;
  `ifndef SYNTHESIS
  begin
   `ifdef ASSERT_MAX_REPORT_ERROR
   error_count = error_count + 1;
   if (error_count <= `ASSERT_MAX_REPORT_ERROR) begin
   `endif
     `ifndef SVA_CHECKER_NO_MESSAGE
     `ifdef SVA_VMM_LOG_ON
     `vmm_error(log,
     $psprintf(
       “SVA_CHECKER_ERROR:%s:%0s:severity %0d: \
        category %0d”,
        msg, err_msg, severity_level, category));
     `else
     $display(
       “SVA_CHECKER_ERROR:%s:%s:%0s:severity %0d: \
        category %0d : time %0t : %m”,
        assert_name, msg, err_msg,
        severity_level, category, $time);
   `endif
```

```
      `ifdef ASSERT_MAX_REPORT_ERROR
    end
    `endif
    `endif
    if (severity_level == 0) sva_checker_finish;
  end
  `endif
  endtask
  ...
  (* category = category *) assert_one_hot:
    assert property (
      @(sampling_ev) disable iff (!not_resetting)
        ($countones(test_expr) == 1) )
      else
        sva_std_error(.err_msg({"failure ", msg});
```

The reporting tasks are included in the *sva_std_task.h* file that is included in each checker. The *vmm_log* class instance and the parameter *category* are visible in the tasks. The user may modify the tasks to suit particular reporting needs.

Rule 3-63 — *Design variables used on the right-hand side of auxiliary state variable assignments shall be sampled with #1step skew.*

Since assertions sample design variables in the pre-poned region, the auxiliary state variables must reflect the same values to present a consistent view of the design state. Therefore, a *clocking* block shall be used to sample these variables with an input skew of *#1step*.

Example 3-46. Sampling of Design Variables for Auxiliary State Variables

```
clocking sampling_ev @(posedge sva_checker_clk);
   input not_resetting, push, push_data, pop, pop_data;
endclocking : sampling_ev
...
// use sampled value in an assignment
always @(sampling_ev) begin
   sva_v_deferred_pop_sr <=
               { sva_v_deferred_pop_sr[pop_lat-2:],
                 sampling_ev.pop};
end
```

Rule 3-64 — *Sequences of events and boolean conditions shall be covered using cover property statements.*

Sequences of events and boolean conditions are most simply and efficiently described using properties. Furthermore, such coverage points are often useful targets for test generation using hybrid-formal tools (provided that the properties are written in a

synthesizable style). Example 3-47 illustrates using *cover property* statements in the *assert_stack* checker to cover an empty stack condition. Similar statements can be written for stack full or high-watermark conditions.

Example 3-47. Cover Property in *assert_stack* for Empty Condition

```
cover_number_of_empty : cover property(
  @(posedge sva_checker_clk)
  (not_resetting) throughout(
     sva_v_deferred_pop ##1
    ( !$stable( sva_v_stack_ptr) &&
      ( sva_v_stack_ptr == 0)) ) );
```

Rule 3-65 — *Data-related coverage shall be implemented using* `covergroup` *constructs.*

Data-related properties usually classify various data values as they are observed into bins, indicating how many times each value or range of values occurred. This classification is usually more efficiently described using the *covergroup* construct. Example 3-48 illustrates this classification on the coverage of various fill levels of the stack in the *assert_stack* checker. Note that, as required by Rule 3-63, it is necessary to introduce sampling of data in the preponed region using the *clocking* construct.

Example 3-48. Coverage of Stack Fill Levels

```
logic [16:0] sva_v_stack_ptr;
always @(sampling_ev) begin
  sva_v_stack_ptr <= ... ;
  past_sva_v_stack_ptr <= sva_v_stack_ptr;
end

...

covergroup depth_cg @(sampling_ev);
  coverpoint sva_v_stack_ptr
    iff (sampling_ev.not_resetting &&
      |(past_sva_v_stack_ptr^sva_v_stack_ptr)) {
        bins observed_depth[] = {[0:depth]};}
        option.per_instance = 1;
        option.at_least = 1;
        option.name = “observed_outstanding_contents”
        option.comment = “Bin index is the fill level”
endgroup : depth_cg

depth_cg depth_cover = new();
```

Suggestion 3-66 — *Coverage of response delays may be implemented using assertion coverage.*

This form of coverage requires support by the underlying simulator. It has to sample the different delays specified in *##* and `*` operators.

Example 3-49. Delay Coverage Using *`cover property`*

```
c1: cover property (
  @(posedge clk) (!reset) throughout
       (!ready_in || accepted_in) ##1
       ready_in ##[1:$] accepted_in );
```

Recommendation 3-67 — *Delay coverage, when delay or latency is measured by counting clock ticks, should be implemented using a* `covergroup` *triggered from a* `sequence.`

Because automatic delay coverage may not be available in the tools, it is preferable to use local variables in a *`sequence`* and a *`covergroup`* for measuring the delays or latencies. This is particularly true when the maximum delay or latency values are large and it can be implemented in a similar way as in Example 3-37 for covering the lengths (duration) of a packet.

Rule 3-68 — *Assertions or coverages that are triggered only at the end of simulation shall be controlled by the macro* *`ASSERT_END_OF_SIMULATION.`*

As specified in the Accellera OVL, the macro *`ASSERT_END_OF_SIMULATION`* can be used to trigger sampling only at the end of simulation. The user must define the macro to be some expression that transitions from zero to one to mark that event. Usually, the sampling clock is also required to trigger the expression. The following example is taken from the *`assert_quiescent`* checker:

Example 3-50. Triggering by *`ASSERT_END_OF_SIMULATION`*

```
`ifdef ASSERT_END_OF_SIMULATION
  (* category = category *) assert_quiescent_state:
  assert property( @( posedge clk)
    disable iff (!not_resetting)
    ($rose(sample_event) ||
     $rose(`ASSERT_END_OF_SIMULATION ==1'b1) )
       |->
         (state_expr == check_value) )
   else sva_checker_error("");
`else
  (* category = category *) assert_quiescent_state:
```

```
  assert property( @( posedge clk)
    disable iff (!not_resetting)
        $rose(sample_event)) |->
                ( state_expr == check_value) )
    else sva_checker_error("");
`endif
```

Rule 3-69 — *Coverage points shall be selectable by parameter(s).*

Once coverage points have been sufficiently covered, it may be desirable to disable them in subsequent runs to speed up the simulation. Also, when some reusable checkers are deployed, not all coverages may be of interest in a given instance of the checker. Therefore, individual coverage points shall be selectable by setting a bit in a parameter of the checker. Each *`cover property`* or *`covergroup`* is associated with one bit in the parameter. By setting the bit to one, a conditional *`generate`* statement will include the coverage into the executable model. In the VMM checkers, there are up to three coverage levels each independently controlled by a parameter. Example 3-51 shows the selection of one cover property at level three, parameter bit position zero.

Example 3-51. Selecting a Coverage Point

```
interface assert_stack (...);
  ...
  parameter coverage_level_3 = 0;
  ...
  generate
    ...
    if( ( coverage_level_3&1 ) != 0)
    begin : cov_level_3_0
      (* category = category, checkerType = "LIMIT",
         checkerLevel = 3 *)
      cover_stack_hi_water_chk : cover property(
        @( posedge sva_checker_clk)
        not_resetting &&
        $rose(sva_v_stack_ptr == hi_water_mark));
    end : cov_level_3_0
    ...
  endgenerate
  ...
endinterface : assert_stack
```

Rule 3-70 — *Checker coverage points shall be subdivided into three levels.*

Level one is associated with the basic coverage representing the triggering condition(s) of the checker and possibly the matching situation.

Level two provides coverage of ranges of values such as data, delay, latency, etc. that require implementation using covergroups.

Level three represents coverage of corner cases of data, latencies, delays, etc., i.e., in general, the limits of the ranges in Level two. These are provided so that the corner case can become a reachability goal for formal tools. Also, in simulation, it may be sufficient to know whether the corner cases have been hit without requiring to know how the ranges in Level two were covered.

Not all levels may have meaning in a particular checker and thus need not always exist.

Rule 3-71 — *Each coverage level shall be controlled by a separate parameter.*

There shall be three parameters controlling the coverage levels independently using the mechanism illustrated in Example 3-51: `coverage_level_1`, `coverage_level_2`, `coverage_level_3`.

Level one coverage must be present in every checker, but the inclusion of the other two levels depends on their usefulness in the particular behavior that is verified by the checker. The default values of the parameters are such that Level one coverage points are the only ones selected.

Assertion-Based Verification IP

This section contains a collection of guidelines for the development of reusable assertion-based verification IP using SystemVerilog. The development of a reusable checkers must take into account many factors, such as: configuration of protocol options, parameterization of signal widths and resource requirements, assertions and/or assumptions, use in simulation and/or formal tools, controls in simulation, reporting of usage errors, failure reporting and message contents, coverage properties and other coverage points, 4-valued and/or 2-valued evaluation of expressions, efficiency, and packaging for SystemVerilog and VHDL design environments.

The guidelines that follow are subdivided into sections that address the above issues based on property rules, architecture and controls, naming convention and coding style, documentation and release items and testing.

The first question that arises when faced with the development of a checker for a bus protocol is often “*Where do I start?*” The protocol description is spread over many pages in different forms: text, timing diagrams, state machines, perhaps some

algorithms and even logic diagrams. For assertion-based verification IP, a list of precise statements that can be expressed in temporal language is required.

Rule 3-72 — *Each rule shall be based on a requirement stated in the specification document.*

The rules shall each have a succinct name, a brief description and a cross-reference to the section (and paragraph number) in the protocol specification document.

The rules should not introduce more constraints on the behavior than the specification imposes because it may result in false failures of assertions. They may also lead to false positive results or inconsistencies if used as assumptions. Overconstraining can happen easily when interpreting timing diagrams in which actual causal relations are not clearly identified. For example, event *B* may be depicted between two to six cycles after event *A*, and event *C* between three to eight cycles after event *A*. Unless a timing constraint between *B* and *C* is also depicted or the textual description implies such a constraint, no temporal relationship between events *B* and *C* should be imposed.

An underconstrained rule will be more permissive and may lead to a false positive outcome if used as an assertion and to a false negative result if used as an assumption. Such a situation may arise when some protocol requirements are spread over multiple diagrams or spread over multiple pages. Underconstraining rules are often best detected if used as assumptions because they may lead to failures on assertions or lead to a dead end in random simulation constrained by these underconstrained assumptions.

Rule 3-73 — *The rules shall be subdivided according to the signal direction they control.*

The distinction of assertions by signal direction is extremely important if the checker is to be used in (hybrid-)formal tools in a system- or a block-level test. In a system-level test, all the rules of the bus are used as assertions. In a block-level test, only those rules that pertain to the block output signals may be used as assertions. The rules that control input signals should be used as assumptions on the inputs of the block. In a simulation tool, all rules are used as assertions.

Often, it may be difficult to identify which signal is to be constrained by a specific rule. Fortunately, with careful reading, the specifications are usually quite clear as to what signal is concerned.

For example, in a single-master single-slave system, there are signals from master to slave (M–S) and from slave to master (S–M). The rules should be subdivided into two subsets according to the direction they characterize, M–S and S–M. Let *ready_in* be a signal from master to slave and *accepted_in* a signal from slave to master. If the rule says "*Whenever ready_in is asserted, then within one to ten cycles accepted_in should be asserted,*" the rule indicates a required behavior of *accepted_in*, assuming that *ready_in* is asserted. Thus, the rule belongs to the S–M subset. This rule could be described by the property shown in Example 3-52.

Example 3-52. Assertion Implementing a Protocol Rule

```
a0: assert property (
  @(posedge clk) disable iff (reset)
  (!ready_in || accepted_in) ##1 ready_in
       |-> ##[1:10] accepted_in )
else $display("...");
```

Note that the predicate condition may refer to signals from either set; it is the signals in the conclusion that determine the subset. Also notice that *ready_in* need not return to zero between requests.

The statement "*When ready_in is asserted, it must remain so until accepted_in is asserted,*" is a rule that controls the behavior of *ready_in* and not that of *accepted_in*. Therefore, the rule belongs to the M–S subset. It can be expressed in a property as shown in Example 3-53.

Example 3-53. Assertion Implementing a Protocol Rule

```
a1: assert property (
  @(posedge clk) disable iff (reset)
  (!ready_in || accepted_in) ##1 ready_in
        |-> ready_in[*1:$] ##0 accepted_in )
```

Notice that there is no time constraint involved. Only the stability of *ready_in* until *accepted_in* is verified. The timing aspect that constrains *accepted_in* is specified in Example 3-52.

The identity of the constrained signal may not be obvious. For example, in the statement "*FRAME must not be asserted if DATA==eof is going to appear on the output within the next four cycles,*" it seems that *FRAME* is to be constrained by some future signal value. However, such a situation is non-causal and can be resolved in two ways. Either there is another signal (possibly internal to the DUT) that does indicate that *DATA==eof* will occur in the future and that signal can be used to constrain *FRAME*, or the constraint is actually on *DATA* not being equal to *eof* if *FRAME* is asserted. The latter case can be expressed as shown in Example 3-54.

Example 3-54. Property Implementing a Protocol Rule

```
property DATA;
   @(clk) FRAME |-> ##1 (DATA != eof)[*4];
endproperty : DATA
```

Suggestion 3-74 — *Rules may refer to internal signals in the device.*

Some rules may be usable only in block-level white-box verification. These rules should be put in a separate subcategory and are not used as assumptions on the design. The inclusion of the rules should be controlled by a parameter of the checker.

Rule 3-75 — *Rules shall refer to internal driver signals of external buses.*

If there are multiple drivers on the same signal within a single design block, it is not possible to identify which driver is currently driving the bus by observing only the signal. By referring to the internal driver enable signal of each driver, it is possible to identify which master is driving the bus and at what point the driving started and stopped.

For example, the *FRAME* signal on a PCI bus is a shared signal among a number of devices. Suppose that the ports of a PCI master checker are *frame_out_n*, *frame_in_n*, and *frame_en_n*. In a white-box verification with access to the internal signals, the checker ports would be mapped to the internal signals:

```
.frame_out_n (DUT_frame_out_n),
.frame_in_n  (DUT_frame_in_n),
.frame_en_n  (DUT_frame_en_n)
```

In a black-box verification in which there is no access to the internal signals *frame_in_n* and *frame_en_n*, the port mapping may be:

```
.frame_out_n (FRAME_n),
.frame_in_n  (FRAME_n),
.frame_en_n  (1'b0)
```

where *FRAME_n* is the shared bus signal.

Recommendation 3-76 —*The set of rules should not contain redundant rules.*

The protocol rules should be disjoint, i.e., a rule should not cover (parts of) behaviors characterized by another rule. The reason is not only the efficiency of the checker, but also easier problem identification in case of a failure of the assertion.

For example, the specification of the PCI protocol contains redundant rules:

Rule 1: If *PERR#* is enabled and a data parity error is detected by a master during a read transaction, the master must assert *PERR#* two clocks after a completion of a data phase in which a parity error occurs. (Section 3.7.4.1 of PCI 2.2 Specification.)

Rule 2: Master always drives *PERR#* (when enabled) for a minimum of one clock for each data phase in which a parity error is detected (Section 3.8.2.1 of PCI 2.2 Specification.)

Rule 1 detects the first occurrence of *PERR#*, and at that point, Rule 2 is also verified because once *PERR#* is asserted, it is for at least one cycle. Rule 2 is thus redundant. In the description, Rule 1 should refer to both items in the protocol specification.

Recommendation 3-77 —*Contradictions in the rules should be eliminated.*

Such contradictions may occur due to misinterpretation of the specification in an area only partially related to another rule. Identifying contradictions is a difficult task. Often, they may be detected only when testing the entire checker and in particular if the properties are used as assumptions.

Rule 3-78 — *Compliance statements or functional coverage shall be expressed using* `cover property` *statements.*

Protocol specifications are often accompanied by compliance statements that state what sequences of transactions or operations must be supported by the design to be compliant. These compliance statements can be expressed using `cover property` statements and used in the evaluation of functional coverage.

Example 3-55. Write Followed by a Read

```
SnpsApb_cv_WriteAfterRead:
   cover property (@(posedge clk)
      SETUP_WR_SELECT ##1 ENABLE_WR_DSEL ##1
      IDLE[*0:$] ##1 SETUP_RD_COV ##1 ENABLE_READ
);
```

Architecture of Assertion-Based IP

When architecting assertion-based verification IP, one must consider various issues: packaging and configuration, protocol layers, reuse of common definitions, `assume` versus `assert`, coverage points, control mechanisms, and failure reporting.

The guidelines specified for the basic checkers in the previous section are applicable to assertion-based verification IP.

Rule 3-79 — *Reusable checkers shall be packaged in an* `interface`*.*

Reusable IP checkers shall be bound or instantiated in a design or on its boundary interfaces. The encapsulation in an `interface` provides the necessary capabilities and implementation flexibility for the internal structure of the IP.

Rule 3-80 — *For use with VHDL testbenches and models, there shall be an equivalent component (entity) declaration.*

The component shall have the same generics as the SystemVerilog interface has parameters and also the same ports as the SystemVerilog interface. For IP that is to be used in mixed SystemVerilog/VHDL environments, the parameter and port types in the parent SystemVerilog checkers should be limited to those available in VHDL and supported by the mixed language tool(s).

Recommendation 3-81 —*Global configuration should be implemented using* `` `define `` *macros.*

Parameters that affect all instances of the checker may be implemented using `` `define `` macros and placed in a header file. This file is then included wherever needed. Only this file needs to be edited when changing the global configuration. The definitions can also be generated randomly by a separate program to build tests in which the architectural configuration is also randomized. An alternative is to use the `+define+` compilation option to specify the actual macro definitions.

Rule 3-82 — *Global coverage and assertions shall be enabled using macros.*

As in the VMM checker library, coverage and assertions should be turned on globally by defining the macro `COVER_ON` and `ASSERT_ON`, respectively.

Rule 3-83 — *Individual coverage points shall be selectable using bits in parameter(s).*

Unlike in the basic checkers, the subdivision into levels may depend on the particular protocol. Therefore, the coverage points must be individually selectable.

Rule 3-84 — *Instance-specific configuration shall be implemented using parameters.*

The majority of configuration values shall be passed to the checker through module or interface parameters because they allow creating multiple instances of the checker each configured differently.

Recommendation 3-85 —*Definitions, sequences, properties and auxiliary variables that are common to different parts or layers of the protocol should be packaged in an* `interface`.

Encapsulating common elements in a separate interface reused in different checkers for the same protocol will ease maintenance. For example, the same definitions may be reused in a checker for the master and for the slave checkers of the PCI protocol.

Rule 3-86 — *For multi-agent protocols properties, each signal direction shall be packaged in a separate top-level* `interface`.

As specified in Rule 3-73, rules shall constrain only signals of one direction. This separation by direction shall also be reflected in the checker architecture. Since there can be multiple instances of each agent-device, the checker for a signal direction may be multiply instantiated with different parameter settings. Example 3-56 shows the interface declaration for a Wishbone bus checker, master and slave.

Example 3-56. Separate Interfaces for Separate Signal Directions

```
(* sva_checker *)
interface wb_master_chk_if (
   CLK_I, RST_I, ACK_I, ADR_O, CYC_O, DAT_I, DAT_O,
   ERR_I, RTY_I, SEL_O, STB_O, WE_O, TAG_O
) ;
... // Body of master protocol checker
    // Verifies behavior of signals driven by master
    // (xyz_O signals)
endinterface : wb_master_chk_if

(* sva_checker *)
interface wb_slave_chk_if (
   CLK_I, RST_I, ACK_O, ADR_I, CYC_I, DAT_I,DAT_O,
   ERR_O, RTY_O, SEL_I, STB_I, WE_I, TAG_O
 ) ;
... // body of slave protocol checker
    // verifies behavior of signals driven by a slave
    // (xyz_O signals)
endinterface : wb_slave_chk_if
```

Rule 3-87 — *Point-to-point protocols properties shall be packaged in a single interface.*

Point-to-point protocols only involve two devices connected over a bundle of wires. A single instance of the checker is required. However, the role of the properties as assumption or assertion must be controlled independently for both signal directions.

Rule 3-88 — *Selection of assumption or assertion role shall be controlled by the parameter <agent>_assume.*

The string *<agent>* identifies the direction of the signals. The parameter *<agent>_assume* selects the directives. When set to one, it shall configure properties to act as assumptions. When set to zero, it shall configure properties to act as assertions.

Example 3-57. Selection of Assume or Assert in MII/MAC Checker

```
// mii_TX_assume is the name of the parameter
`ifdef ASSERT_ON
  generate
    if(mii_RX_assume) begin : mii_RX_assumptions
      always @(rxc) begin
        mii_RX_2: assume property(mii_2_p);
        mii_RX_7: assume property(mii_RX_7_p);
      ...
    end : mii_RX_assumptions

    else begin : mii_RX_assertions
      always @(rxc) begin
        mii_RX_2: assert property(mii_2_p)
        else begin
          sva_checker_error("...");
          RX_SetFrameError(MII_RX_CRS_NOT_W_COL);
        end
        mii_RX_7: assert property(mii_RX_7_p)
        else begin
          sva_checker_error("...");
          RX_SetFrameError(MII_RX_BAD_OR_NO_SDF);
        end
        ...
    end : mii_RX_assertions
  endgenerate
'endif
```

Rule 3-89 — *Single-layer protocols shall use a two-level or a flat architecture.*

Many on-chip and external memory or system bus protocols have a simple one-layer structure. Most signaling is achieved out-of-band using separate control signals. The preferred architecture reflects the flat format of the protocol specification.

Shared items between a number of assertions/coverage items, such as common sequence and property definitions and static auxiliary variable declaration and

assignments may be placed in an *interface*. The *interface* is instantiated and the shared items are referenced through the interface instance name.

Example 3-58. A Two-Level Architecture

```
interface itf_common(logic reset,
                     logic clk,
                     ...);
... // sequences, properties and auxiliary variables
    // shared by a number of assertions / coverage items
endinterface

interface VMM_protocol_master // Can also be interface
    #( ... )// parameters
     ( /// ports
     input logic reset,
     input logic clk,
     input logic sig_a,
     input logic sig_b,
      ...        );  // other ports
  itf_common common_items;

  (* category = assert_category *)
    a: assert property(
        common_items.p1(sig_a) );
    ... // other assertions and coverage items
endinterface
```

Rule 3-90 — *Multi-layer protocols shall use a multi-level architecture.*

In multi layered protocols, like PCI Express, the checks are usually at different abstraction levels in each layer. Therefore, it is preferable to implement each protocol layer checks as a separate section or level in the checker, similar to the layer architecture shown in Figure 4-5. For complex protocols, each layer may be packaged in a separate *interface*. These levels are placed in the top-level *interface* of the checker for each signal direction. The communication link between the levels should be implemented as a well defined block of code that performs the abstraction function between protocol layers. Again, for complex protocols, this block of code could be encapsulated in an *interface* to hide implementation details. For example, the communication link may contain byte counters and packet assembly and disassembly statements that relate a byte-level protocol layer with a packet layer. The layered structure simplifies modeling because the rules and encapsulation follow the protocol definition. Furthermore, the communication link code controls the direction of signal flow depending on the use context: inward for assertions and outward toward the design ports for assumptions.

For example, consider the following protocol. Device *Dev1* asserts *req* and holds until *ack* is received from *Dev2*. At that point, data is placed by *Dev2* on signal *data* and is valid for one clock cycle—the duration of *ack*, which is also one cycle wide. *req* must stay high until and including *ack* is asserted, and then *req* must be deasserted for at least one cycle. *ack* cannot be asserted unless *req* is asserted. Upon reception of two bytes of data, the sixteen bits are XORed. The result must be equal to zero.

A multi-layer protocol checker, shown in Example 3-59, has the following structure. Because the protocol is quite simple and point-to-point, a single top-level *interface* named *DL_PL_checker* is used. It contains the code for the physical layer checks—*PL*—the data link layer checks—*DL*—and the communication link. The communication link does the packet assembly from *data* bytes into a variable called *packet* for assertions on data or disassembly from *packet* into *data* bytes for assumptions on data.

The selection of *assert* or *assume* is done by the parameters *dl_pl_ack_assume* and *dl_pl_req_assume*. When *dl_pl_ack_assume* is set to one, the checker acts as an assumption on *ack* and *data*. When *dl_pl_req_assume* is set to zero, the checker acts as assumption on *req*. Otherwise the checker implements assertions.

Example 3-59. Multi-Level Protocol Checker Architecture

```
// Two-level data link - physical layer checker
// Packets of length 2 bytes only in this example

interface DL_PL_checker
  #(parameter bit dl_pl_ack_assume = 0,
              dl_pl_req_assume = 0)
  (input rst, clk, req, ack, inout [7:0] data);

// physical layer checks PL  // properties of req
  property p_req1; // stable req
    @(posedge clk) disable iff (rst)
      $rose(req) |-> req[*1:$] ##0 ack;
  endproperty : p_req1

  property p_req2; // req Return-to-Zero
    @(posedge clk)  disable iff (rst)
      ack |-> ##1 !req;
  endproperty : p_req2

  property p_req3;
    @(posedge clk)  disable iff (rst)
      ack |-> ##[1:6] req;
```

```
    endproperty : p_req3

  // properties of ack
    property p_ack1; // pulse only
      @(posedge clk) disable iff (rst)
        ack |-> ##1 !ack;
    endproperty : p_ack1

    property p_ack2; // no ack w/o req
      @(posedge clk) disable iff (rst)
        !req |-> !ack;
    endproperty : p_ack2

    property p_ack3; // ack latency
      @(posedge clk) disable iff (rst)
        $rose(req) |-> ##[1:5] ack;
    endproperty : p_ack3

    generate

      if (dl_pl_ack_assume) begin : drive_ack
        aa1: assume property (p_ack1);

        aa2: assume property (p_ack2);

        aa3: assume property (p_ack3);
      end : drive_ack

      else begin : verify_ack
        aa1: assert property (p_ack1) else
          $display("ack too wide");

        aa2: assert property (p_ack2) else
          $display("spurious ack");

        aa3: assert property (p_ack3) else
          $display("ack too late");
      end : verify_ack

      if (dl_pl_req_assume) begin : drive_req
        ar1: assume property (p_req1);

        ar2: assume property (p_req2);

        ar3: assume property (p_req3);
      end : drive_req

      else begin : verify_req
        ar1: assert property (p_req1) else
          $display("req dropped");
```

```
      ar2: assert property (p_req2) else
        $display("req not returned to 0");

      ar3: assert property (p_req3) else
        $display("req too late");
    end : verify_req

  endgenerate

// Communication link with DL layer

// variables for the DL layer
  logic [1:0][7:0] packet = 0;
  logic cnt = 0; // only 2 octets

  clocking p_clk  @(posedge clk);
    input rst, ack, data;
  endclocking : p_clk

// count 2 data octets
  always @(p_clk)
    cnt <= rst ? 0 : p_clk.ack ?
                       !cnt : cnt;
  generate

    if (dl_pl_ack_assume) begin : drive_pl
// disassemble packet into octets
      assign data = rst ?
        0 : ack ?
        (cnt ? packet[1] : packet[0]) :
        0;

// assure stable packet during disassembly
      assume_data_2: assume property
        ( disable iff (rst)
          (1'b1 ##1 !$fell(cnt))
            |-> $stable(packet)
         );
    end : drive_pl

    else begin : verify_dl
//assemble packet for DL layer assertions
      always @(p_clk)
        packet[cnt] <= p_clk.rst ?
           0 : p_clk.ack ?
                 p_clk.data :
                  packet[cnt];
    end : verify_dl
```

```
  endgenerate

// data link layer check DL

  property p_data_1;
    @(posedge clk)
    disable iff (rst)
      (1'b1 ##1 $fell(cnt))
         |-> ((^packet) == 0);
  endproperty : p_data_1

  generate

    if (dl_pl_ack_assume) begin : drive
// assumption on data
      adt1: assume property (p_data_1);
    end : drive

    else begin : verify
// assertion on data
      adt1: assert property (p_data_1) else
                  $display("data_property failed");
    end : verify

  endgenerate

endinterface : DL_PL_checker
```

Rule 3-91 — *Users shall have control over individual assertions at run time.*

Assertion names must be listed in the documentation so that the user can enable/disable individual checks using system tasks at run time.

Rule 3-92 — *Users shall be able to enable or disable assertions or coverage of the entire checker at run time.*

It must be possible to globally enable or disable all assertions or assumptions or coverage using system tasks, using either the system tasks `$assertoff` and `$asserton` with the hierarchical name of the checker instance or using a `category` associated with the checker.

Rule 3-93 — *Formal subset selection shall be controlled using the* `SVA_CHECKER_FORMAL` *macro.*

The control is the same as in the basic VMM checkers, as described in Example 3-61.

Rule 3-94 — *Each assertion shall produce a short but meaningful message with useful variable values upon failure using the message service.*

This is the same output reporting as in VMM checkers, as specified in Rule 3-62.

Rule 3-95 — *Parameter values shall be verified for valid values and value combinations.*

The validity of parameter values shall be verified in an initial block and the simulation shall be terminated with failure indication if they are inconsistent.

Rule 3-96 — *Coverage of variable values shall be implemented using* `covergroups`.

As in VMM checkers, coverage shall be implemented using both `cover property` and coverage groups depending on the type of the data covered. However, for use with formal tools, it is preferable to implement the corner cases using `cover property`.

Recommendation 3-97 —*Checkers shall include an* `assert_category` *parameter.*

The parameter should be assigned to the category attribute of all assertions.

Recommendation 3-98 —*Checkers shall include a* `cover_category` *parameter.*

The parameter should be assigned to the category attribute of all cover property statements and `covergroup` instances.

If the tools support this functionality, these two parameters permit controlling and interrogating the assertions and cover points according to the category value, the same way as in the basic checkers.

Documentation and Release Items

Rule 3-99 — *The release of an assertion-based verification IP shall contain the following items:*

- Files containing the checker modules/interfaces
- Header file with user-configured macro definitions
- All the tests used for verifying the IP

- README file describing the contents of the release
- Text file with Release notes indicating any restrictions related to this particular release not documented in the main User Guide
- User Guide

Rule 3-100 — *A User Guide for the assertion-based verification IP shall include the following information:*

- Checker documentation version and date
- Protocol version and specifications covered by the checker
- Summary of features relative to configuration, protocol subsets, etc.
- Simulation controls
- Coverage information
- Table of rules, separated by signal direction/device type
- List of macro definitions if any, indicating the default behavior and permissible values
- Description of ports and parameters including defaults for each checker unit
- List of any restrictions and limitations of the checker
- Global organization of the checker (details as far as allowed by required IP protection rules, if any)
- Layout of the directory structure in the checker distribution
- Example of use, including sample binding statements in block-level tests under assertion and assumption forms and in system-level assertion-only form
- Instructions on how to compile and execute tests and how to interpret the results
- Information on which assumptions are sufficient to prove individual assertions.

QUALIFICATION OF ASSERTIONS

Assertions in general and reusable checkers in particular have to be thoroughly verified for compliance with their requirements. Bugs can arise due to improper implementation of the assertion itself, misinterpretation of the assertion language constructs, incorrect parameter values and port connections in the bind statements of checkers, typographical errors, or ambiguity in the protocol specification. An error in an assertion or in the deployment of a checker may result in false reporting of successful design verification. The qualification problem is complex because, not

only the acceptance of correct protocol behavior has to be verified, but also that violations of the specification must be detected and reported by the assertion or checker. In other words, both the good and the wrong behaviors must be thoroughly verified.

In the case of reusable protocol checkers, the space of possible violations may be quite large and a carefully prepared and reviewed verification plan is an absolute necessity for assuring acceptable quality of the checker. There are a number of ways to verify that a given assertion or checker complies with the requirements. Depending on the complexity of the checked behavior, the methods vary from simple to very complex involving hybrid formal tools.

Visual inspection — This inspection may be sufficient if the abstraction level of the assertion matches the natural language specification. Visual inspection requires solid understanding of assertion language semantics. Typically, this is used on simple assertions involving a few temporal operators.

For example, the *`until`* property:

```
c |-> d[*1:$] ##1 e;
```

It is sufficient to make sure that the boolean expressions *`c`*, *`d`* and *`e`* are correct., and that the corner timing cases are valid.

In-situ — Qualification in the context of the DUT is the easiest method to implement, but may only exercise a subset of all accepting sequences. In addition, error injection into the design requiring temporary modifications is necessary to verify that the assertion rejects incorrect behaviors.

Assertion testbenches — A testbench can be written to exercise the assertion or the reusable checker, like any other DUT. For any realistic protocol checker, the verification development represents a major endeavor and should not be underestimated.

Formal verification — An interesting alternative may be to use the assertions in the checker as assumptions in a system model configured only using a set of wires and continuous assignments to the bus. The formal tool is used to generate random test sequences that satisfy these assumptions and observe them in a waveform viewer to check their validity. This approach will generate only the “positive” tests. In a subsequent step, *`cover property`* statements are added that encode the negations of the assertions. These *`cover`* statements are then used as goals in the formal tool to ascertain that none of these goals are reachable with the set of assertions used as assumptions. Additional *`cover property`* statements that represent the positive

corner cases of each behavioral rule are then used as search goals to ascertain that all of them are reachable. If not, these corner cases cannot be reached.

The amount of work using the formal approach may not be any less than writing a functional testbench. However, the advantage is that it will also verify that the set of assertions are not mutually contradictory and that they can be used effectively as assumptions. The limitation of this approach may be the performance limits of the tool on complex multi-layered protocols. Also, the boolean expressions and auxiliary variable assignments must be synthesizable.

SUMMARY

In this chapter, assertions and their usage in the design verification process were introduced first. This section was followed by guidelines for inserting assertions inside the DUT by the designers, and then for using assertions on external interfaces as applied by the design verification personnel. In the latter case, the DUT is viewed as a black box.

The rest of the chapter concentrated on providing guidance on good coding practices for assertions and coverage, and for constructing assertion-based checkers and verification IP. This content included a description of a number of standard VMM macro symbols and parameters used in VMM-compatible checkers. The objective was also to create verification IP that can be used alone or integrated with a testbench monitor. Such an integration is possible due to the powerful features of the SystemVerilog language.

The guidelines were presented with simulation in mind as the primary verification tool. However, at the block level, assertions with formal and emulation tools are gaining on importance; therefore, the information in this chapter was written so as not to contradict the constraints imposed by these tools. The more strict rules to satisfy such constraints are the subject of Chapter 7.

For further reading on SystemVerilog Assertions:

Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, "*SystemVerilog Assertions Handbook … for Formal and Dynamic Verification"*, Vhdlcohen Publishing.

Srikanth Vijayaraghavan, Meyyappan Ramanathan, "*A Practical Guide for SystemVerilog Assertions"*, Springer.

CHAPTER 4 TESTBENCH INFRASTRUCTURE

One of the challenges when transitioning from a procedural language, such as Verilog or VHDL, to a language with object-oriented features—such as SystemVerilog—is making effective use of the object-oriented programming model. When used properly, these features can greatly enhance the reusability of testbench components. This section contains guidelines or directives to make the most out of these language features to create verification components and combine them into a powerful verification environment that will satisfy the needs of all the testcase that must be applied to the DUT.

This chapter will be of interest to those responsible for creating a verification environment for a particular design. It will also be of interest to those creating reusable verification IP. The requirements for the transactors and the verification environment that use them will be defined by the verification planning process (Chapter 2) and the selected stimulus and response checking mechanisms (Chapter 5). They must also offer the appropriate data sampling interfaces to allow the implementation of functional coverage points (Chapter 6).

The guidelines presented in this chapter are based on the VMM Standard Library specified in Appendix A. Although the methodology and approaches described here could be implemented in a different class library, using a well-defined and openly accessible library guarantees interoperability of the various verification components.

TESTBENCH ARCHITECTURE

This section describes the recommended testbench architecture. Testcases are implemented on top of a verification environment, as illustrated in Figure 4-1. The verification environment implements the abstraction and automation functions that help minimize the number and details of testcases that need to be written. The verification environment will also be reused, without modifications, by as many testcases as possible to minimize the amount of code required to verify the DUT. For a given design under test, there may be several verification environments as illustrated in Figure 8-4. But the number of environments should be minimized and testcases built on top of existing environments as much as possible. Minimizing the number of lines required to implement a testcase is an important objective of this methodology. Investing in the few—or one—verification environments to save a single line in the potentially thousands of testcases will be a worthwhile investment.

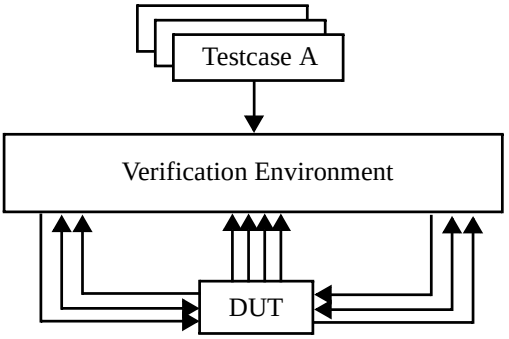


Figure 4-1. Tests on Top of Verification Environment.

Verification environments are not monolithic. As illustrated in Figure 4-2, environments are composed of layers, mirroring the abstraction layers in the data processed by the design—as shown in Figure 4-3—and to meet the varied requirements of all the testcases that need to be written on top of it. Each layer provides a set of services to the upper layers or testcases, while abstracting it from the lower-level details.

Although Figure 4-2 shows testcases interacting only with the upper layers of the verification environment, they can by-pass any layer to interact with any component of the environment or the DUT to accomplish their goal. Testcases are implemented as a combination of additional constraints on generators, new random scenario definitions, synchronization mechanisms between transactors, error injection enablers, DUT state monitoring and directed stimulus. Because the verification environment must be able to support, without modification, all testcases required to

verify the DUT, it must be assembled with carefully designed, reusable components. The next sections will describe how to build such components.

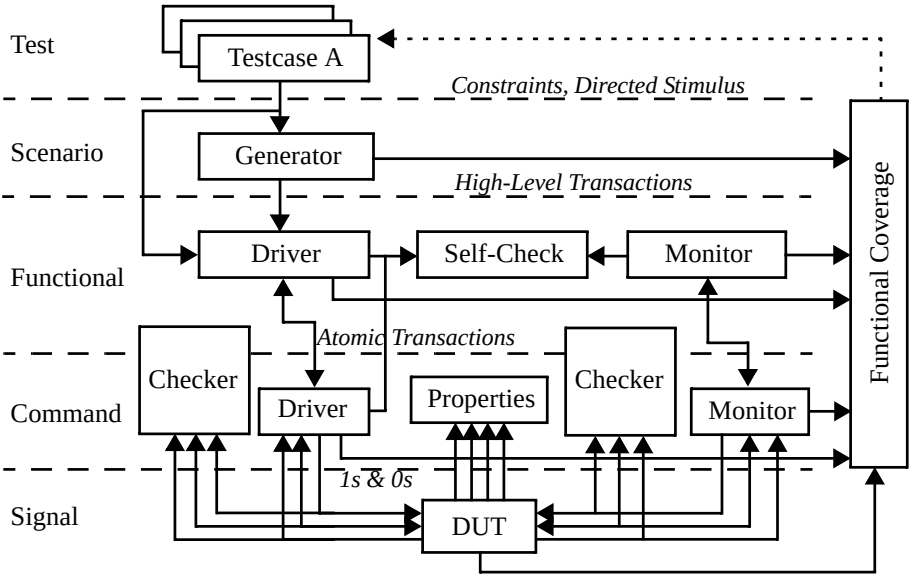


Figure 4-2. Layered Verification Environment Architecture.

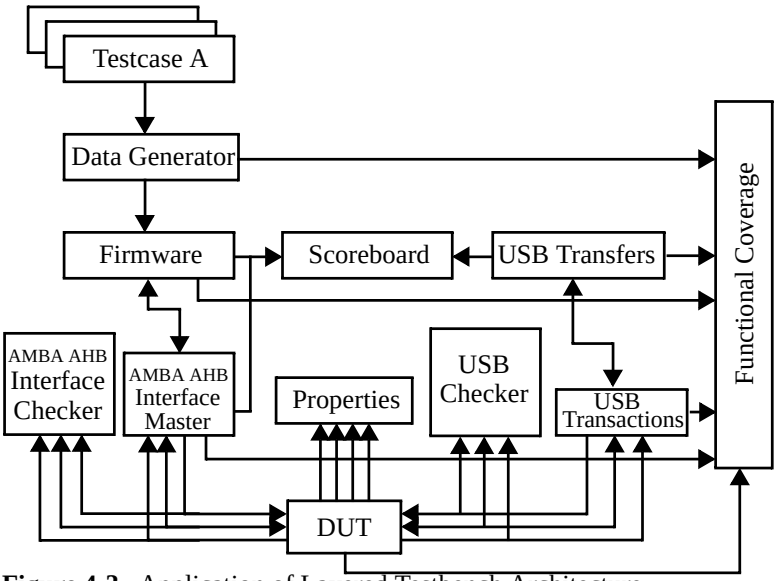


Figure 4-3. Application of Layered Testbench Architecture

Complete verification environments are never implemented in one shot. They are not delivered to the testcase writers as a finished product from the ivory tower of verification architects. Rather, they evolve to meet the increasingly complex requirements of the testcases being written and responses being checked. Initially supporting only a trivial directed testcase with no self-checking, layers are added to evolve them into full-fledged self-checking constrained-random verification environment. The methodology described in this chapter allows this evolution to occur in a backward-compatible fashion to avoid breaking existing testcases.

The layered architecture makes no assumption about the DUT model. It can be an RTL or gate-level model as well as a transaction-level model. The DUT can also be simulated natively in the same simulator as the verification environment, co-simulated on a different simulator or emulated on a hardware platform.

Figure 4-4 shows the general structure used to implement complete testbenches.

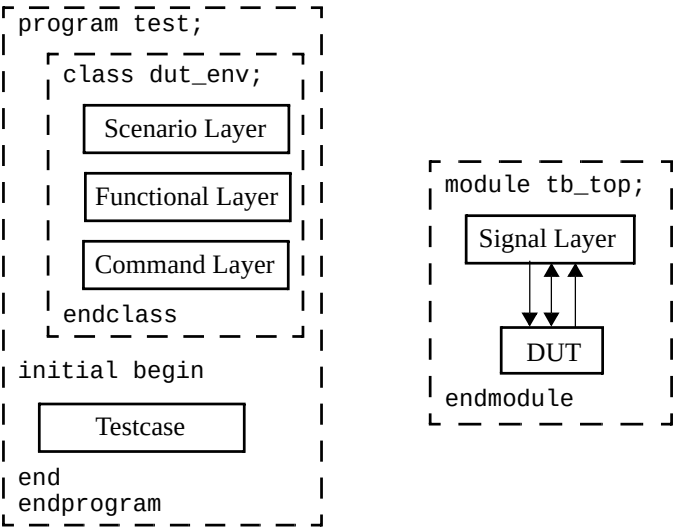


Figure 4-4. General Implementation Structure

Rule 4-1 — *The signal layer shall be implemented and the DUT shall be instantiated in a top-level module*

This top-level module will contain the design portion of the simulation. Any element of the signal layer or DUT will be accessible via cross-module references through the

top-level module. There is no need to instantiate the top-level module anywhere. Guidelines for implementing the signal layer can be found in “Signal Layer” on page 107.

Rule 4-2 — *The verification environment shall be implemented in a top-level class*

The environment will leverage generic functionality from a verification environment base `class`. It will be able to refer to the signal layer or any DUT element via cross-module references into the top-level module. The environment will be instantiated by each testcase. Guidelines for implementing the top-level environment class can be found in “Simulation Control” on page 124.

Rule 4-3 — *Testcases shall be implemented in an initial block in a program block*

The `initial` block describes the testcase procedure. The testcase procedure can refer to any public members in the verification environment via hierarchical references through the top-level environment `class`. It will also be able to access any element of the signal layer or DUT via cross-module references through the top-level module. The verification environment will be designed to minimize the number of statements in testcases. Guidelines for implementing testcases can be found in “Test Layer” on page 123 and in “Generating Stimulus” on page 211.

Signal Layer

This layer provides signal-level connectivity to the DUT. The signal layer provides pin name abstraction to enable verification components to be used, unmodified, with different DUTs or different implementation models of the same DUT—for example, an RTL description of the DUT using `interface` constructs and a gate-level description of the same DUT using individual bit I/O signals. This layer may also abstract synchronization and timing of synchronous signals with respect to a reference signal.

The signal abstraction provided by this layer is accessible and may be used by all layers and testcases above it, if signal-level access is required. However, verification environments and testcases should be implemented in terms of the highest possible level services provided by lower layers and avoid accessing signals directly unless absolutely necessary.

Rule 4-4 — *`interfaces` shall be packaged in the same file as the transactors that use them.*

Command-layer transactors have a physical-level interface composed of individual signals. All signals pertaining to a physical protocol are bundled in a single `interface` construct. The `interface` declarations and the `class` declarations for transactors operating on those interfaces are intimately related. They shall be packaged together in the same file.

Example 4-1. Packaging of `interface` Declaration

```
interface mii_if(...);
   ...
endinterface: mii_if;
...
class mii_phy_layer ...;
   virtual mii_if.phy_layer sigs;
   ...
endclass: phy_layer
...
```

If an `interface` declaration already exists for the protocol signals—for example in the RTL design code—and it meets (or can be made to meet) all of the subsequent requirements outlined in this section, then it should be physically moved to the file packaging the transactors that will use them. In most cases, different `interface` declarations will exist or be required. Their respective signals can be mapped to each other as described in Rule 4-14 on page 113.

Rule 4-5 — *`Interfaces` shall be named with a prefix that matches the associated component name prefix or `package` name.*

To minimize the collisions between `interface` names and other identifiers in the global name space, they shall use a likely-unique prefix. That prefix should be the same as any prefix used for related transactors. The name of the `package` that optionally contains the transactors that make use of the `interface` may also be used as the prefix to further document the association.

Rule 4-6 — *All interface signals shall be declared as `wire`.*

The same interface constructs are used by verification components regardless of their perspective or role on the interface. Some components will drive signals, others will simply monitor their value. Which signal is being driven or monitored may be

different depending on the functionality of the verification component. Wires are able to represent a physical interface signal regardless of the direction of the signal.

Example 4-2. Verification `interface` Signal Declaration

```
interface mii_if();
   wire       tx_clk;
   wire [3:0] txd;
   wire       tx_en;
   wire       tx_err;
   wire       rx_clk;
   wire [3:0] rxd;
   wire       rx_dv;
   wire       rx_err;
   wire       crs;
   wire       col;
   ...
endinterface: mii_if
```

Rule 4-7 — *Synchronous interface signals shall be sampled and driven using a* `clocking` *block.*

This approach will avoid race conditions between the design and the verification environment, and it will allow the verification environment to work with RTL and gate-level models of the DUT without any modifications or timing violations.

Example 4-3. Synchronous Interface Signal Declaration

```
interface mii_if;
   ...
   parameter setup_time = 5ns;
   parameter hold_time  = 3ns;
   clocking mtx @(posedge tx_clk);
      default input #setup_time output #hold_time;
      output txd, tx_en, tx_err;
   endclocking: tx

   clocking mrx @(posedge rx_clk);
      default input #setup_time output #hold_time;
      input rxd, rx_dv, rx_err;
   endclocking: rx
   ...
endinterface: mii_if
```

Furthermore, code located in program blocks executes in the *reactive* region. This execution implies that all pending signal assignments will have been made to their target variables. Any variable assigned using a non-blocking assignment sampled after a clock edge would have its current value sampled, not its previous value. In

Example 4-4, the module would report a value of `Q` equal to 0; whereas, the program would report a value of `Q` equal to 1. Using a clocking block is the only way to sample the value present in a variable before the clock transition.

Example 4-4. Sampling Differences Between Module and Program Blocks

```
module dff;

reg Q = 0;
reg clk = 0;
initial #10 clk = 1;

always @ (posedge clk) Q <= 1;

always @ (posedge clk) $write("Module Q = %b\n", Q);

endmodule: dff

program p;
initial
   forever begin
      @ (posedge dff.clk) begin
         $write("Program Q = %b\n", dff.Q);
      end
   end
endprogram
```

Rule 4-8 — *Set-up and hold time in clocking blocks shall be defined using parameters.*

This style will allow changing the set-up and hold time, on a per instance basis, to meet the needs of the DUT without having to modify the interface declaration itself. Modifying the interface declaration would have global effects; whereas, parameters can be specified for each interface instance.

Example 4-5. Specifying Set-Up and Hold Times for Synchronous Signals

```
mii_if #(.setup_time(1),
         .hold_time  (0)) mii();
```

Rule 4-9 — *Individual `modports` shall be declared for each type of proactive, reactive and passive transactors.*

Different transactors may have different perspectives on a set of signals. For example, one may be a master driver, another a reactive monitor or a slave driver and another may be a passive monitor. Certain interfaces may have different types of proactive transactors, such as arbiters and agents. To ensure that each transactor uses the

interface signals appropriately, a *modport* must be declared for each of their individual perspectives.

Example 4-6. Module Port Declarations

```
interface mii_if;
   ...
   modport mac_layer(clocking mtx,
                     clocking mrx,
                     input    crs,
                     input    col, ...);
   ...
   modport phy_layer(clocking ptx,
                     clocking prx,
                     output   crs,
                     output   col, ...);
   ...
   modport passive(clocking ptx,
                   clocking mrx,
                   input crs,
                   input col, ...);
   ...
endinterface: mii_if
```

As described in “Transactors” on page 161, transactors will be implemented as separate *class* definitions and will interface to the physical signals through *virtual modports*. Transactions and transactors should not be defined as *tasks* inside the *interface* declaration. An *interface* declaration that is shared with the RTL design may contain such *tasks*, but they should not be used by the verification environment.

Rule 4-10 — *The direction of asynchronous signals shall be specified in the* *modport* *declaration*

As per Rule 4-6, the signals declared in the interface create a bundle of wires. The direction of information on the individual wires depends on the role of the agent connected to those wires. For example, wires carrying address information will be outputs for a bus master but will be inputs for a bus slave or bus monitor. The direction of asynchronous signals is specified directly in the *modport* as they are not sampled via *clocking* blocks.

Rule 4-11 — *The direction of synchronous signals shall be specified in the* `clocking` *block declaration.*

The direction of synchronous signals is specified in the clocking block. As per Rule 4-12, the entire clocking block is included in the modport port list. Thus, the synchronous signals are already visible and their direction are already enforced.

This rule implies that they will generally be one `clocking` block per `modport` for each clock domain in the `interface`

Rule 4-12 — *The* `clocking` *block shall be included in* `modports` *port list instead of individual clock and synchronous signals.*

Transactors must be able to delay the driving or sampling of synchronous signals by an integer number of cycles. By referring to the clocking block that defines the synchronization of an interface, an integer number of cycles can be specified without having to know the details of the synchronization event that is specified in the `clocking` block declaration. Because all signals in a `clocking` block are visible, adding the synchronous signals to the `modport` port list would be redundant. Furthermore, having to refer to synchronous signals through their respective `clocking` blocks highlights their synchronous nature and associated sampling and driving semantics.

Example 4-7. Waiting for the Next Cycle on the `tx` Interface

```
foreach (bytes[i]) begin
   ...
   @(this.sigs.mtx);
   this.sigs.mtx.txd <= nibble;
   ...
   @(this.sigs.mtx);
   this.sigs.mtx.txd <= nibble;
   ...
end
```

Rule 4-13 — *The design and all required interfaces and signals shall be instantiated in a* `module` *with no ports.*

This top-level module contains both the verification environment and the design. Since the verification environment is responsible for the stimulus and monitoring of all signals to and from the DUT, this top-level module does not require any further inputs or outputs. It will also minimize the amount of code that will be duplicated in

all of the tests when instantiating this top-level module, since no ports will need to be mapped.

Example 4-8. Simple Top-Level Module

```
module tb_top;

bit tx_clk;
bit rx_clk;
...
mii_if #(...) mii();

assign mii.tx_clk = tx_clk;
assign mii.rx_clk = rx_clk;
...
eth_top    dut(...
               .mtx_clk_pad_i(mii.tx_clk),
               .mtxd_pad_o   (mii.txd),
               ...);
...
endmodule: tb_top
```

Rule 4-14 — *Signals in different* `interface` *instances implementing the same physical interface shall be mapped to each other.*

Verification components and the design may have been written using different `interface` declarations for the same physical signals. To connect the verification components to the design, it will be necessary to map two separate `interface` instances to the same physical signals. This can be accomplished with continuous assignments for unidirectional signals and aliasing for bidirectional signals.

Example 4-9. Mapping Two Different Interface Instances to the Same Physical Signals

```
interface eth_tx_if;   // RTL Design Interface
   bit         clk;
   wire  [3:0] d;
   logic       en;
   logic       err;
   logic       crs;
   logic       col;
endinterface: eth_tx_if

module tb_top;

bit        tx_clk;
eth_tx_if  mii_dut(); // Design Interface Instance
mii_if     mii_xct(); // Transactor Interface Instance
```

```
assign mii_dut.clk    = tx_clk;  // Unidirectional
assign mii_xct.tx_clk = tx_clk;
alias  mii_xct.txd    = mii_dut.d; // Inout
...
endmodule: tb_top
```

Rule 4-15 — *Clock generation shall be done in the top-level module.*

Clock signals must be scheduled in the design regions. They must therefore be generated outside of the verification environment, in an *always* or *initial* block. Clock signals should not be generated inside verification components or transactors because they are scheduled in the reactive region.

Rule 4-16 — *There shall be no clock edges at time 0.*

There are race conditions between initial scheduling of the *initial* and *always* blocks implementing the clock generators and those implementing the design. Delaying the clock edges to a point in time until after all *initial* and *always* blocks have had the chance to be scheduled at least once eliminates those race conditions.

It is a good idea to wait for the duration of a few periods of the slowest clock in the system before generating clock edges.

Example 4-10. Clock Generation in Top-Level Module

```
module tb_top;
bit tx_clk;
...
initial
begin
   ...
   #20; // No clock edge at T=0
   tx_clk = 0;
   ...
   forever begin
      #(T/2) tx_clk = 1;
      #(T/2) tx_clk = 0;
   end
end
endmodule: tb_top
```

Rule 4-17 — *The `bit` type shall be used for all clock and reset signals.*

Using a two-state type ensures that the clock signals will be initialized to a known, valid value.

If a four-state logic type, such as `logic`, is used to implement the clock signals, the initialization of those signals to `1'b0` may be considered as an active negative edge by some design components. The alternative of leaving the clock signals at `1'bx` while the clock edges are being delayed—as per the previous rule—may cause functional problems if these unknown values are propagated.

Recommendation 4-18 —*The timing relationship of unrelated clock signals should be randomized as part of the testcase configuration.*

Clock signals with an asynchronous relationship are inherently synchronized when simulated with a fixed initial phase and a common timing reference such as the internal simulation time. To ensure that problems related to asynchronous clock domains can surface during simulation, the relationship of such clocks should be randomized.

Example 4-11. Randomizing Clock Offsets

```
integer tx_rx_offset;  // 0-99% T lag
integer T = 100;
initial
begin
   ...
   tx_rx_offset = {$random} % 100;
   #20; // No clock edge at T=0
   tx_clk = 0;
   rx_clk = 0;
   ...
   fork
      begin
         #(T * (tx_rx_offset % 100) / 100.0);
         forever begin
            #(T/2) rx_clk = 1;
            #(T/2) rx_clk = 0;
         end
      end
   join_none

   forever begin
      #(T/2) tx_clk = 1;
      #(T/2) tx_clk = 0;
   end
end
```

To enable tests to control the random clock relationship values, these values should be randomized as part of the testcase configuration descriptor. The randomized value would then be assigned to the appropriate variable in the clock generation code, in the extension of the `vmm_env::reset_dut()` method. See “Simulation Control” on page 124.

Command Layer

Transaction-level testcases written using Verilog or VHDL are typically implemented on top of a command layer.

The command layer typically contains bus-functional models, physical-level drivers, monitors and checkers associated with the various interfaces and physical-level protocols present in the DUT. The command layer provides a consistent, low-level transaction interface to the DUT, regardless of how the DUT is modeled. At this level, a transaction is defined as an atomic data transfer or command operation on an interface, such as a register write, the transmission of an Ethernet frame, or the fetching of an instruction. Atomic operations are typically defined using individual timing diagrams in interface specifications.

Reading and writing registers is an example of an atomic operation. The command layer provides methods to access registers in the DUT. To speed up device initialization, this layer may have a mechanism that bypasses the physical interface to peek and poke the register values directly into the DUT model. Such choice should be selectable at run time, where all subsequent register accesses would be done in the same manner until the mode selection is modified. Note that the implementation of a direct-access, register read/write driver is dependent upon the implementation of the DUT.

A driver actively supplies stimulus data to the DUT. A *proactive* driver is in control of the initiation and type of the transaction. Whenever a new transaction is supplied by the higher layers of the verification environment to a proactive driver, the transaction is immediately executed on the physical interface. For example, a master bus-functional model for an AMBA AHB interface is a proactive driver.

A *reactive* driver is not in control of the initiation or type of the transaction, but may be in control of some aspect of the timing of its execution, such as the introduction of wait states. The transaction is initiated by the DUT, and the reactive driver supplies the required data to successfully complete the transaction. For example, a program memory interface bus-functional model is a reactive driver: The DUT initiates *read* cycles to fetch the next instruction and the bus-functional model supplies new data in the form of an encoded instruction.

A monitor reports observed high-level transaction timing and data information. A *reactive* monitor includes elements to generate the required low-level handshaking signals to terminate an interface and successfully complete a transaction. Unlike a reactive driver, a reactive monitor does not generate transaction-level information. For example, a Utopia Level 1 receiver is a reactive monitor: it receives ATM cells without having to generate additional data but generates a *cell enable* signal back to the DUT for flow control. A *passive* monitor simply observes all signals involved in the transaction without any interference. A passive monitor is suitable for monitoring transactions on an interface between two DUT blocks in a system-level verification environment.

When interfacing with an RTL or gate-level model, the physical abstraction layer may translate transactions to or from signal assertions and transitions. When interfacing with a transaction-level model, the physical abstraction layer becomes a pass-through layer. In both cases, the transaction-level interface presented to the higher layers remains the same, allowing the same verification environment and testcases to run on different models of the DUT, at different levels of abstraction, without any modifications.

The services provided by the command layer may not be limited to atomic operations on external interfaces around the DUT. These services can be provided on internal interfaces for missing—or temporarily removed—design components. For example, an embedded memory acting as an elastic buffer for routed data packets could be replaced with a testbench component to help track and check packets in and out of the buffer rather than only at DUT endpoints. Or, an embedded code memory in a processor could be replaced with a reactive driver that would allow on-the-fly instruction generation instead of using pre-loaded static code. Alternatively, an embedded processor could be replaced with a transactor to allow the testbench to control the read and write cycles of the processors instead of indirectly through code execution. When replacing DUT components with a transactor, care must be taken that it be configured to an equivalent functionality. For example, if the transactor implements a superset of the transactions or timing compared to the DUT component, it should be configured to restrict its functionality to match that of the DUT component.

Rule 4-19 — *Drivers and monitors shall execute in the reactive region.*

This implementation will ensure that they avoid race conditions between the design, the assertions and the verification environment. If Rule 4-22 and Rule 4-89 are followed, this rule will be automatically followed.

Example 4-12. Low-Level Driver Implemented in the Top-Level Module file

```
module tb_top;
...
bit rst = 0;
...
endmodule: tb_top
...
program tb_top_env;

   task reset;
      tb_top.rst <= 1;
      ...
   endtask: reset

endprogram: tb_top_env
```

Rule 4-20 — *Drivers and monitors shall be implemented as transactors.*

Although a different label is used to refer to command-layer transactors from functional-layer transactors, they only differ in their interfaces. The high-level interface on both sets of transactors is a transaction-level interface. The low-level interface on command-layer transactors is a physical-level interface and is designed to connect to the DUT. On a functional-layer transactor, the low-level interface is a transaction-level interface, which is designed to connect to the high-level interface of another transactor.

In all other respects, command- and functional-layer transactors operate in the same way and should be implemented using the same techniques and offer the same type of capabilities.

Recommendation 4-21 —*A monitor transactor should be configurable as reactive or passive.*

Instead of having two separate monitors, it will be easier to maintain a configurable monitor because the functionality can be shared between the two flavors. Furthermore, environments built on top of a reactive monitor can be reused in a system-level environment with little modification by reconfiguring the reactive monitor into a passive mode.

Functional Layer

The functional layer provides the necessary abstraction layers to process application-level transactions and verify the correctness of the DUT.

Unlike interface-based transactions of the physical layer, the transactions in the functional layer may not have a one-to-one correspondence with an interface or physical transaction. Functional transactions are abstractions of the higher-level operations performed by a major subset of or the entire DUT, beyond the physical interface module. A single functional transaction may require the execution of dozens of command-layer transactions on different interfaces. It may depend on the completion status of some physical transactions to retry some transactions or delay others.

Functional layer transactors can be proactive, reactive or passive. A proactive transactor controls the initiation and the kind of transaction and typically supplies some or all of the data required by the transaction. A reactive transactor does not control the initiation nor the kind of transaction, and it is only responsible for terminating the transaction appropriately by supplying response data or handshaking. Reactive transactors may report the observed transaction data they are reacting to. Passive transactors monitor transactions on an interface and simply report the observed transactions.

At all times, the self-checking structure included in this layer verifies the correctness of the response of the DUT, based on the configuration and stimulus streams. The correctness may be determined at various levels of abstraction—physical or functional— according to the functionality being verified. The correctness of the response should not imply or require that a particular model of the DUT is used, nor should it depend on unspecified ordering or timing relationships among the transactions.

As illustrated in Figure 4-5, the functional layer should be sub-layered according to the protocol structure. For example, a functional layer for a TCP/IP over Ethernet device should contain a sub-layer to transmit—and if necessary retry—an Ethernet frame. Additional sub-layers may be provided to encapsulate IP fragments into Ethernet frames, fragment large IP frames into smaller IP fragments that will fit into a single Ethernet frame and encapsulate a TCP packet into an IP frame.

A testcase is performed as a series of functional transactions at the appropriate level of abstraction, rather than always using low-level physical transactions or physical signals.

It must be possible to turn off all or some of the higher sub-layers. As tests are implemented, they are first concerned with verifying the lower-level operations of the DUT. These lower-level operations correspond to the lowest sub-layers of the functional layer. Stimulus—and self-checking—is performed at the relevant

abstraction sub-layer to easily create the relevant scenarios and corner cases for that level of abstraction.

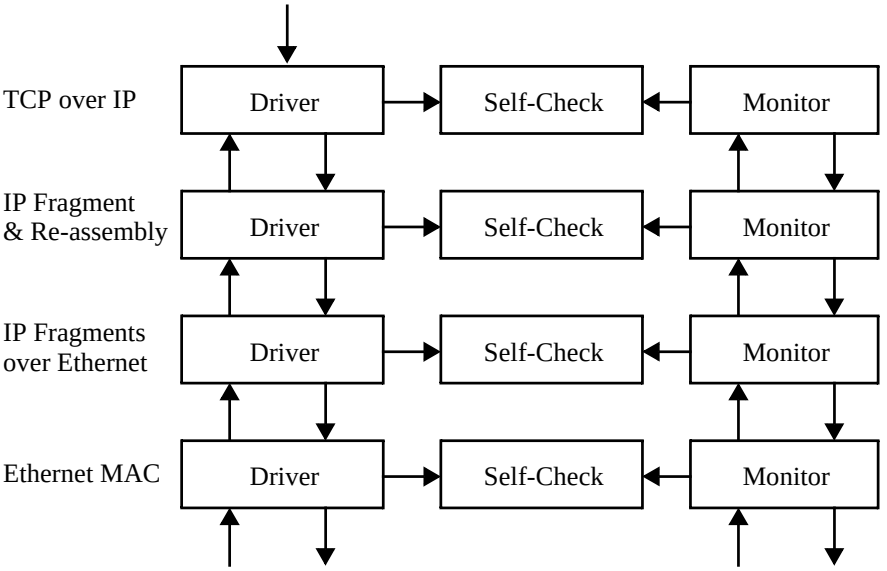


Figure 4-5. Functional Sub-Layers

This checking requires that any stimulus provided by the higher sub-layers be turned off to prevent undesirable noise from affecting a testcase. This requirement is often a by-product of the implementation of the verification environment itself: It is typically implemented bottom-up, with the low-level testcases implemented first. As additional levels of functionality are being verified, additional sub-layers are added to the functional layer. To maintain backward compatibility with the existing lower-level testcases, these additional sub-layers must be disabled by default.

The functional layer is also responsible for configuring the DUT according to a configuration descriptor. The configuration descriptor is a high-level description of the DUT configuration that is “compiled” into the necessary register reads and writes and embedded memory images.

This layer includes a functional coverage model for the high-level stimulus and response transactions. It records the relevant information on all transactions processed or created by this layer.

Rule 4-22 — *Transactors shall execute in the reactive region.*

This implementation will ensure that they avoid race conditions between the design, the assertions and the verification environment.

If Rule 4-91 is followed, transactors are implemented using *classes*. Classes instantiated in *program* blocks and whose threads are started in *program* blocks will have reactive semantics, which is a consequence of Rule 4-28.

Example 4-13. Transactors Instantiated in *program* block

```
class mii_phy_layer extends vmm_xactor;
   ...
endclass: mii_phy_layer
...
class tb_env extends vmm_env;
   ...
   mii_phy_layer phy;
   ...
   virtual function void build();
      ...
      this.phy = new(...);
      ...
   endfunction: build
   ...
   virtual task start();
      ...
      this.phy.start_xactor();
      ...
   endtask: start
endclass: tb_env

program test;
   tb_env env = new;
   ...
endprogram
```

Rule 4-23 — *Monitors shall be implemented as transactors.*

Although a different label is used to refer to stimulus transactors (driver) from response transactors (monitor), they only differ in the direction of the information flow. The interfaces on both sets of transactors are transaction-level interfaces. In all other aspects, drivers and monitors operate in the same way and should be implemented using the same techniques and offer the same type of capabilities.

Recommendation 4-24 —*A reactive transactor should be configurable as passive.*

Instead of having two separate transactors, it will be easier to maintain a configurable transactor because the functionality can be shared between the two flavors. Furthermore, environments built on top of an reactive transactor can be reused in a system-level environment with little modification by reconfiguring the reactive transactor into a passive mode.

A reactive functional layer driver will usually have a request/response transaction-level interface, as described in section titled "Reactive Response" on page 192. When configured as a passive monitor, the direction of the response channel is simply reversed to observe the response generated from a previous request.

Scenario Layer

This layer provides controllable and synchronizable data and transaction generators. By default, they initiate broad-spectrum stimulus to the DUT. Different generators or managers are used to supply data and transactions at the various sub-layers of the functional layer. This layer also contains a DUT configuration generator.

Atomic generators generate individually constrained transactions. Atomic generators are suitable for generating stimulus where putting constraints on sequences of transactions is not necessary. For example, the configuration description generator is an atomic generator.

Scenarios are sequences of random transactions with certain relationships. Each scenario represents an interesting sequence of individual transactions to hit a particular functional corner case. For example, a scenario in an Ethernet networking operation would be a sequence of frames with a specified density—i.e., a certain portion of the time the Ethernet line is busy sending/receiving, and otherwise, the line is idle. Scenario generators generate scenarios in random order and sequence and produce a stream of transactions that correspond to the generated scenarios. Scenario managers initiate scenarios as defined by and under the direction of a particular testcase and produce a stream of transactions that corresponds to the requested scenarios.

This layer may be partially or completely bypassed by the test layer above it, depending on the amount of directedness required by the testcase. Consequently, generators must be able to be turned off, either from the beginning or in the middle of a simulation, to allow for the injection of directed stimulus. The generator must also be able to be restarted to resume the generation of random stimulus after a directed stimulus sequence.

Rule 4-25 — *Generators shall execute in the reactive region.*

This implementation will ensure that they avoid race conditions between the design, the assertions and the verification environment. This rule is a consequence of Rule 4-26 and a special case of Rule 4-22.

Rule 4-26 — *Generators shall be implemented as transactors.*

A generator is a transactor with an output transaction-level interface and usually without any input interfaces. In all other aspects, generators behave like transactors and should be implemented using the same techniques and offer the same type of capabilities.

Example 4-14. Generator are Transactors

```
class eth_frame_gen extends vmm_data;
   ...
endclass
```

Test Layer

Testcases involve a combination of modifying constraints on generators, the definition of new random scenarios, synchronization of different transactors and the creation of directed stimulus.

This layer may also provide additional testcase-specific self-checking not provided by the functional layer at the transaction level. For example, checks where correctness will depend on timing with respect to a particular synchronization event introduced by the testcase.

Rule 4-27 — *Testcases shall be implemented in a single* `initial` *block in a* `program` *block.*

This implementation will ensure that they are executed as part of the reactive region and avoid race conditions among the design, the assertions and the verification environment. Using a single `initial` block limits the execution of a testcase to a single overarching thread, making it easier to understand the steps involved in accomplishing that testcase. This rule is consistent with Rule 4-3.

Rule 4-28 — *Testcases shall instantiate the verification environment in a* `program` *block.*

The environment instantiates all necessary transactors and manages their execution. Rule 4-22 requires that transactors execute in the reactive region. Therefore, the

environment that encapsulates them must be instantiated in a *program* block. As an added benefit, the *program* block implementing the testcase will be able to access any required element of the verification environment. The environment is instantiated in a local variable to prevent initialization race conditions.

Example 4-15. Testcase Accessing Verification Environment Elements

```
program test;
...
tb_env env = new;
initial
begin
   env.run();
end
endprogram: test
```

Rule 4-29 — *Testcases shall access elements in the top-most module or design via absolute cross-module references.*

The *program* block implementing the testcase must be able to access any required element of the signal layer or the design. The top-most module is never instantiated and its elements are accessed via the *$root* name space.

Example 4-16. Testcase Accessing Design Elements

```
program test;
...
initial
begin
   fork
      forever begin
         @ (posedge tb_top.wb_int);
         `vmm_note(env.log, "Interrupt asserted!");
         ...
      end
   join_none
   ...
end
endprogram: test
```

SIMULATION CONTROL

In general, the successful simulation of a testcase to completion involves the execution of the following major functions:

1. Generating the testcase configuration.

   This generation includes a description of the verification environment configuration and the DUT configuration. This generation also includes a description of the testcase duration. It is used by the self-checking structure to determine the appropriate response to expect and by the verification environment to configure the DUT.

2. Building the verification environment around the DUT according to the generated testcase configuration.

   The specific type and number of transactors that need to be instantiated around the DUT to exercise it correctly may depend on the configuration that will be used. For example, a DUT may be configured with an Intel-style or a Motorola-style processor interface. Each will require a different command-layer transactor. Similarly, 16 GPIO pins may be configured as 16 1-bit interfaces or one 16-bit interface (or anything in between). Each configuration will require a different number of command-layer and functional-layer transactors and scoreboards in the self-checking structure.

3. Disabling all assertions and resetting the DUT.

4. Configuring the DUT according to the generated testcase configuration.

   This configuration may involve writing specific values to registers in the DUT or setting interface pins to specific levels.

5. Enabling assertions and starting all transactors and generators in the environment.

   Transactors and generators should not be started as soon as they are instantiated. The DUT must first be configured to be ready to correctly receive any stimulus. Starting the generators too soon complicates the response checking because some initial stimulus sequences must be ignored.

6. Detecting the end-of-test conditions.

   The end of a test may be determined using a combination of conditions. Depending on the DUT, a testcase is terminated after running for a fixed amount of time or number of clock cycles or number of transactions or until a certain number of error messages have been issued or when all monitors are idle.

7. Stopping all generators in an orderly fashion.

8. Draining the DUT and collecting statistics.

   To determine success of a simulation, it may be necessary to drain the DUT of any buffered data or download accounting or statistics registers. Any expected data left in the scoreboard is then assumed to have been lost. The value of accounting or statistics registers is compared against their expected values.

9. Reporting on the success or failure of the simulation run.

OOP Primer: Virtual Methods

Virtual methods behave differently from non-virtual methods when their implementation is overloaded in a class extension. A base class *bc* contains a virtual method *vm()* and a non-virtual method *nvm()*. A derived class *dc* overloads both of these methods:

```
class bc;                       class dc extends bc;
   virtual task vm();              virtual task vm();
      $display("bc::vm()");           $display("dc::vm()");
   endtask                         endtask
   task nvm();                     task nvm();
      $display("bc::nvm()");          $display("dc::nvm()");
   endtask                         endtask
endclass: bc                    endclass: dc
```

Code calling those methods will invoke different implementations depending on the type of class handle used to refer to the method.

```
initial begin
   dc dc_obj = new;      // Both handles refer to same
   bc bc_obj = dc_obj;   // instance of class "dc"

   dc_obj.vm();          // -> $display("dc::vm()");
   dc_obj.nvm();         // -> $display("dc::nvm()");
   bc_obj.vm();          // -> $display("dc::vm()");
   bc_obj.nvm();         // -> $display("bc::nvm()");
end
```

Not all DUTs require all of those steps. Some steps may be trivial for some DUTs. Others may be very complex. But every successful simulation follows this sequence of generic steps. Individual testcases intervene at various points in the simulation flow to implement the unique aspect of each testcase.

As illustrated in Figure 4-6, the *vmm_env* base class (See “vmm_env” on page 365.) formalizes these simulation steps into well-defined *virtual methods*. These methods must be extended for a verification environment to implement its DUT-specific requirements. The *vmm_env* base class supports the development of a verification environment by extending each virtual method to implement the individual simulation steps as required by the target DUT. The base class already contains the functionality to manage the sequencing and execution of the simulation steps. The DUT-specific environment class extension also instantiates and interconnects all

transactors, generators and self-checking structures to create a complete layered verification environment around the DUT.

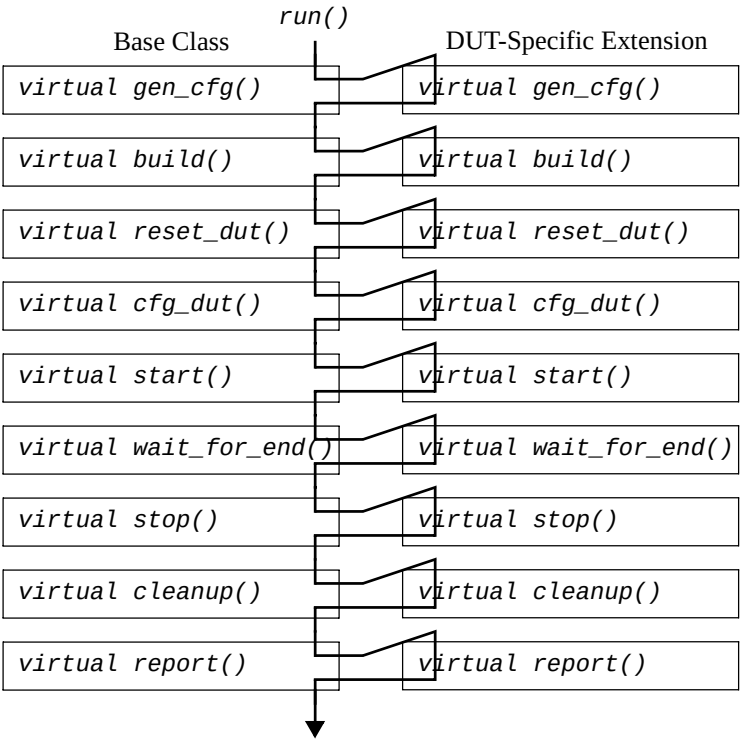


Figure 4-6. Execution Sequence in *vmm_env* Class

Note that the simulation sequence does not allow a testcase to invoke the *reset_dut()* method in the middle of a simulation—i.e., during the execution of *wait_for_end()*. It is often necessary to verify that the design can be properly dynamically reset and reconfigured. The body of the *vmm_env::reset_dut()* and *vmm_env::cfg_dut()* should be implemented in separate tasks. A hardware reset testcase would simply call these tasks directly to perform the hardware reset and reconfiguration. The reset and reconfiguration sequence would be considered part of the *wait_for_end* step for that particular testcase, not a separate step in the simulation.

Rule 4-30 — *Extensions of the* `vmm_env` *class shall implement the* `gen_cfg()`, `build()`, `reset_dut()`, `cfg_dut()`, `start()`, `wait_for_end()`, `stop()` *and* `cleanup()` *virtual methods.*

These methods implement each of the generic steps that must be performed to successfully simulate a testcase. They must be overloaded to perform each step as required by the DUT. Even if a method does not need to be extended for a particular DUT, it is should be extended anyway—and left empty—to explicitly document that fact.

Rule 4-31 — *Extensions of the* `gen_cfg()`, `build()`, `reset_dut()`, `cfg_dut()`, `start()`, `wait_for_end()`, `stop()` *and* `cleanup()` *virtual methods shall call their base implementation first.*

The implementation of these methods in the base class manages the sequence in which these methods must be invoked. They make it unnecessary for each testcase to enumerate all intermediate simulation steps. To ensure the proper automatic ordering of the simulation steps, each method extension must call their base implementation first.

If this rule is violated, the execution sequence of the various simulation steps will be broken.

Example 4-17. Extending Simulation Step Methods

```
class tb_env extends vmm_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      ...
   endtask
   ...
endclass
```

Rule 4-32 — *Extensions of the* `vmm_env` *class shall not redefine the* `vmm_env::run()` *method.*

This method is not virtual because it is not intended to be specialized for a particular verification environment. It is the method that executes the virtual methods in the proper sequence. It must not be redefined to prevent its semantics from being modified. See “task run()” on page 366.

Rule 4-33 — *The extension of the* `vmm_env::gen_cfg()` *method shall randomize the testcase configuration descriptor using a factory pattern.*

This extension will let tests constrain the testcase configuration descriptor to ensure that a desirable configuration is generated without requiring modifications to the environment or configuration descriptor. See “Controlling Random Generation” on page 227 for more details on the factory pattern. The randomized configuration value can be further modified procedurally once this method returns.

Example 4-18. Randomization of Testcase Configuration Descriptor

```
class tb_env extends vmm_env;
   test_cfg cfg;
   ...
   function new();
      super.new();
      this.cfg = new;
      ...
   endtask

   virtual function void gen_cfg();
      super.gen_cfg();
      if (!this.cfg.randomize()) ...
   endfunction: gen_cfg
   ...
endclass: tb_env
```

The testcase configuration descriptor includes all configurable elements of the DUT and the execution of a testcase. Not only does it describe the various configurable features of the design, but also it may include simulation parameters such as asynchronous clock offsets, as required by Recommendation 4-18. It may also include other variable parameters such as how long to run the simulation for, how many instances of the DUT in the system or the MAC addresses of “known” external devices.

Some environments may not have any randomizable parameters. Albeit rare, these environments have a configuration that is described by an empty configuration descriptor, a descriptor without any `rand` class properties or a descriptor constrained to a single solution.

Rule 4-34 — *Environment components shall be instantiated only in the* `vmm_env::build()` *method extension.*

Transactors, generators, scoreboards and functional coverage models must be instantiated according to the testcase configuration, which is final only when the `vmm_env::build()` method is invoked. No environment components must be instantiated in the constructor or other methods. The only object that is instantiated in the environment constructor is the default testcase configuration descriptor instance that will be a randomized `vmm_env::gen_cfg()` method extension.

Example 4-19. Instantiating Environment Components

```
class tb_env extends vmm_env;
   ...
   function new();
      super.new();
      this.cfg = new;
      ...
   endtask
   ...
   virtual function void build();
      super.build();
      ...
      this.phy_src = new("Phy Side", 0);
      ...
   endfunction: build
   ...
endclass: tb_env
```

Rule 4-35 — *All transactors and generators shall be instantiated in public class properties.*

A testcase needs to be able to control transactors and generators as required to implement the objectives of the testcase. The transactors and generators can be controlled directly if they are publicly accessible.

Example 4-20. Transactor Properties in Verification Environment

```
class tb_env extends vmm_env;
   ...
   eth_frame_atomic_gen host_src;
   eth_frame_atomic_gen phy_src;
   eth_mac              mac;
   mii_phy_layer        phy;
   ...
endclass: tb_env
```

Rule 4-36 — *Self-checking integration callback instances shall be registered in the* `vmm_env::build()` *method extension.*

Integrating the scoreboard into the environment is part of the building process. All necessary references will exist and can be passed to the callback extension constructors if required. See “Integration with the Transactors” on page 253 for more details.

Example 4-21. Integrating Scoreboard Via Callbacks

```
class tb_env extends vmm_env;
   ...
   virtual function void build();
      ...
      begin
         sb_mac_sbc sb = new(...);
         this.mac.append_callback(sb);
      end
      ...
   endfunction: build
   ...
endclass: tb_env
```

Rule 4-37 — *The self-checking integration callbacks shall be the first to be registered with a transactor.*

Additional callback extensions can be registered with the same transactor using the `vmm_xactor::append_callback()` or `vmm_xactor::prepend_callback()` method. They can be used to sample data into a functional coverage model or modify the data for error injection. The self-checking structure must be aware of all known exceptions or errors that were injected in the stimulus or observed on the response to be able to correctly predict the expected response or assess the correctness of the observed response.

By registering the scoreboard integration callbacks first, they can act as a reference point for subsequent callback registrations. Any callback that can modify the callback information is registered before it—so they will be invoked before the scoreboard integration callbacks. Other callbacks are registered afterward, thus being called after the scoreboard integration callback, ensuring that they observe the same data that was used to determine the correctness of the response.

Rule 4-38 — *Callback extension instances that can modify or delay the transactions shall be registered before the scoreboard callback extension instances.*

Some callback extensions may modify or delay the transaction before it is processed by the transactor. For example, error injection callback extensions could corrupt a parity byte. Registering those callback extensions before the scoreboard callback extensions ensures that the scoreboard will see the actual transaction that will be executed.

Because the scoreboard callback extensions are registered first, these callback extensions are registered using the `vmm_xactor::prepend_callback()` method.

Rule 4-39 — *Callback extension instances that do not modify the transactions shall be registered after the scoreboard callback extension instances.*

Some callback extensions do not modify the transaction and simply record its content. For example, functional coverage callback extensions may save some transaction parameters for later sampling by a coverage group. Registering those callback extensions after the scoreboard callback extensions ensures that the content of the transaction that was checked for correctness will be sampled.

Because the scoreboard callback extensions are registered first, these callback extensions are registered using the `vmm_xactor::append_callback()` method.

Recommendation 4-40 —*The `vmm_env::cfg_dut()` method should have a fast implementation that writes to registers and memories via direct accesses.*

Configuring a DUT often takes a significant amount of simulation time because a relatively slow processor or serial interface is usually used to perform the register and memory updates. Once that interface has been verified to ensure that all registers and memories can be updated, it is no longer necessary to keep exercising that logic.

The DUT-specific extension of the `vmm_env::cfg_dut()` method should have a “fast-mode” implementation, controlled by a parameter in the testcase configuration descriptor, that causes all register and memory updates to be performed via direct or API accesses, bypassing the normal processor interface.

Rule 4-41 — *The extension of the `vmm_env::start()` method shall start all transactors and generators.*

The environment should not require any additional external intervention to operate properly. All transactors and generators must be started in the extension of the *`vmm_env::start()`* method. If a testcase does not require the presence or operation of a particular transactor, it can be stopped afterwards.

Example 4-22. Starting Transactors

```
class tb_env extends vmm_env;
   ...
   virtual task start();
      super.start();
      ...
      this.mac.start_xactor();
      ...
   endtask: start
   ...
endclass: tb_env
```

Configuring the DUT often requires that the configuration and host interface transactors be started in the extension of the *`vmm_env::cfg_dut()`* method.

Rule 4-42 — *The `vmm_env::wait_for_end()` method shall have configurable aspects.*

A testcase must be able to control how long it is going to run. It may be in terms of number of transactions to be executed or absolute time. There must be some properties in the testcase configuration descriptor, used by the *`vmm_env::wait_for_end()`* method, that control the duration of a simulation.

Example 4-23. Configurable Testcase Duration

```
class tb_env extends vmm_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      ...
      wait (this.cfg.run_for_n_tx_frames == 0 &&
            this.cfg.run_for_n_tx_frames == 0);
      ...
   endtask: wait_for_end
   ...
endclass: tb_env
```

Rule 4-43 — *The `vmm_env::wait_for_end()` method shall return when the `vmm_env::end_test` event is triggered.*

The *`vmm_env::end_test`* event can be used by directed testcases or other functions of the verification environment to indicate that the end-of-test condition has been detected.

Example 4-24. Extension of the *`vmm_env::wait_for_end()`* Method

```
class verif_env extends vmm_env;
   ...
   virtual task wait_for_end();
      super.wait_for_end();
      fork
         ... // Other termination conditions
         @(this.end_test) disable wait_for_end;
      join_none
      ...
   endtask: wait_for_end
   ...
endclass: tb_env
```

Suggestion 4-44 — *The `vmm_env::wait_for_end()` method may have other termination conditions.*

End-of-test conditions are entirely user-defined. The occurrence of any of the end-of-testcase conditions detected by the method extension must cause the method to return.

Message Service

Transactors, scoreboards, assertions, environment and testcases use messages to report any definite or potential errors detected. They may also issue messages to indicate the progress of the simulation or provide additional processing information to help diagnose problems.

To ensure a consistent look and feel to the messages issued from different sources, a common message service should be used. A message service is only concerned with the formatting and issuance of messages, not their cause. For example, the time reported in a message is the time at which the message was issued, not the time a failed assertion started. The VMM message service uses the following concepts to describe and control messages: source, filters, type, severity and handling. See the section titled "vmm_log" on page 368 for more details.

Message Source — Each instance of the message service interface object represents a message source. A message source can be any component of a testbench: a command-layer transactor, a sub-layer of the self-checking structure, a testcase, a

generator, a verification IP block or a complete verification environment. Messages from each source can be controlled independently of the messages from other sources.

Message Filters — Filters can prevent or allow a message from being issued. Filters are associated and disassociated with message sources. They are applied in order of association and control messages based on their identifier, type, severity or content. Message filters can promote or demote messages severities, modify message types and their simulation handling. After a message has been subjected to all the filters associated with its source, its effective type and severity may be different from the actual type and severity originally specified in the code used to issue a message.

Message Type — Individual messages are categorized into different types by the author of the code used to issue the message. Assigning messages to their proper type lets a testcase or simulation produce and save only (or all) messages that are relevant to the concerns addressed by a simulation. Table 4-1 summarizes the available message types and their intended purposes:

Table 4-1. Message Types

Message Type	Purpose
vmm_log::FAILURE_TYP	An error has been detected. The severity of the error is categorized by the message severity.
vmm_log::NOTE_TYP	Normal message used to indicate the simulation progress.
vmm_log::DEBUG_TYP	Message used to provide additional information designed to help diagnose the cause of a problem. Debug messages of increasing detail are assigned lower message severities.
vmm_log::TIMING_TYP	A timing error has been detected (e.g., set-up or hold violation).
vmm_log::XHANDLING_TYP	An unknown or high-impedance state has been detected or driven on a physical signal.

Table 4-1. Cont.

Message Type	Purpose
vmm_log::REPORT_TYP *vmm_log::PROTOCOL_TYP* *vmm_log::TRANSACTION_TYP* *vmm_log::COMMAND_TYP* *vmm_log::CYCLE_TYP*	Additional message types that can be used by transactors.
vmm_log::INTERNAL_TYP	Messages from the VMM base classes. Should not be used when implementing user-defined extensions.

Message Severity — Individual messages are categorized into different severities by the author of the code used to issue the message. A message's severity indicates its importance and seriousness and must be chosen with care. For fail-safe reasons, certain message severities cannot be demoted to arbitrary severities. Table 4-2 summarizes the available message severities and their meaning:

Table 4-2. Message Severities

Message Severity	Indication
vmm_log::FATAL_SEV	The correctness or integrity of the simulation has been compromised. By default, simulation is aborted after a fatal message is issued. Fatal messages can only be demoted into error messages.
vmm_log::ERROR_SEV	The correctness or integrity of the simulation has been compromised, but simulation may be able to proceed with useful result. By default, error messages from all sources are counted and simulation aborts after a certain number are observed. Error messages can only be demoted into warning messages.
vmm_log::WARNING_SEV	The correctness or integrity of the simulation has been potentially compromised, and simulation can likely proceed and still produce useful result.
vmm_log::NORMAL_SEV	This message is produced through the normal course of the simulation. It does not indicate that a problem has been identified.

Table 4-2. Cont.

Message Severity	Indication
vmm_log::TRACE_SEV	This message identifies high-level internal information that is not normally issued.
vmm_log::DEBUG_SEV	This message identifies medium-level internal information that is not normally issued.
vmm_log::VERBOSE_SEV	This message identifies low-level internal information that is not normally issued.

Simulation Handling — Different messages require different action by the simulator once the message has been issued. Table 4-3 summarizes the available message handling and their default trigger:

Table 4-3. Simulation Handlings

Simulation Handling	Action
vmm_log::ABORT_SIM	Terminates the simulation immediately and returns to the command prompt, returning an error status. This is the default handling after issuing a message with a *vmm_log::FATAL_SEV* severity.
vmm_log::COUNT_ERROR	Count the message as an error. If the maximum number of such messages from all sources has exhausted a user-specified threshold, the simulation is aborted. This is the default handling after issuing a message with an *vmm_log::ERROR_SEV* severity.
vmm_log::STOP_PROMPT	Stop the simulation immediately and return to the simulation runtime-control command prompt.
vmm_log::DEBUGGER	Stop the simulation immediately and start the graphical debugging environment.
vmm_log::DUMP_STACK	Dump the callstack or any other context status information and continue the simulation.
vmm_log::CONTINUE	Continue the simulation normally.

The guidelines below are to be applied when creating messages from within transactors, data and transaction models, the self-checking structure, the verification environment itself or testcases.

Rule 4-45 — *All simulation messages shall be sent through the message service.*

Do not use *`$display()`* to manually produce output messages. If a predefined method that produces output text must be invoked (such as the *`vmm_data::psdisplay()`* method), do so within the context of a message.

Example 4-25. Issuing a Message with Externally-Displayed Text

```
vmm_log log = new(...);
...
if (log.start_msg(vmm_log::DEBUG_TYP,
                  vmm_log::TRACE_SEV)) begin
   log.text("Transmitting frame...");
   log.text(fr.psdisplay("   "));
   log.end_msg();
end
```

Rule 4-46 — *Messages of type `FAILURE_TYP` shall be of severity `WARNING_SEV`, `ERROR_SEV` or `FATAL_SEV` only.*

A failure of lower severity does not make sense, except when being demoted to prevent its issuance.

Recommendation 4-47 —*Messages of type `FAILURE_TYP` should be issued using the `‘vmm_warning()`, `‘vmm_error()` or `‘vmm_fatal()` macros.*

These macros provide a shorthand notation for issuing single-line failure messages.

Example 4-26. Using a Macro to Issue a Message

```
‘vmm_error(this.log, "Unable to write to TxBD.TxPNT");
```

Rule 4-48 — *Messages of type `NOTE_TYP` shall be of severity `NORMAL_SEV` only.*

A note of higher or lower severity does not make sense, except when being demoted to prevent its issuance or promoted to detect unexpected code execution.

Recommendation 4-49 —*Messages of type `NOTE_TYP` should be issued using the `‘vmm_note()` macro.*

This macro provides a shorthand notation for issuing single-line note messages.

Rule 4-50 — *Messages of type `DEBUG_TYP` shall be of severity `TRACE_SEV`, `DEBUG_SEV` or `VERBOSE_SEV` only.*

A debug message of higher severity does not make sense, except when being promoted to detect unexpected code execution.

Recommendation 4-51 —*Messages of type `DEBUG_TYP` should be issued using the `‘vmm_trace()`, `‘vmm_debug()` or `‘vmm_verbose()` macros.*

These macros provide a shorthand notation for issuing single-line debug messages.

Recommendation 4-52 —*Calls to text output tasks should be made only once it has been confirmed that a message will be issued.*

The `$display()`, `$write()` and `$sformat()` system tasks are runtime expensive. These tasks are also heavily used in the implementation of any `vmm_data::psdisplay()` method extension. They should be called only when their formatted output will be required. Example 4-25 creates the formatted output only if the message will be issued.

VCS provides the `$psprintf()` function that returns the formatted string instead of writing it into a string, like `$sformat()` does. This function can be used with the message macros, to display messages with runtime formatted content. The macros are designed to invoke the `$psprintf()` function only if the message will be issued, as per this recommendation.

Example 4-27. Using a Macro and the `$psprintf()` System Function

```
`vmm_debug(this.log,
           $psprintf("Buffering TX Frame at 'h%h:\n%s",
                     tx_pnt, fr.psdisplay("   ")));
```

DATA AND TRANSACTIONS

One of the challenges when transitioning from a procedural language, such as Verilog or VHDL, to an object-oriented language such as SystemVerilog, is making effective use of the object-oriented programming model. This section contains guidelines or directives to help strike the right balance between objects and procedures.

Rule 4-53 — *A data item shall be modeled using a* `class`*, not a* `struct` *nor a* `union`*.*

A data item is any atomic amount of data eventually or directly processed by the DUT. Packets, instructions, pixels, picture frames, SDH frames and ATM cells are all examples of data items. A data item can be composed of smaller data items by composing a class from smaller classes. For example, a class modeling a picture frame could be composed of thousands of instances of a class modeling individual pixels.

Example 4-28. Ethernet MAC Frame Data Model

```
class eth_frame extends vmm_data;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand bit [15:0] len_typ;
   rand bit [ 7:0] data[];
   rand bit [31:0] fcs;
   ...
endclass: eth_frame
```

Using the `class` construct has advantages over using `struct` or `union` constructs. The latter would only be able to model the values contained in the data item; whereas, classes can also model operations and transformations—such as calculating a CRC value or comparing two instances—on these data items using `methods`. `Class` instances are more efficient to process and move around as only a reference to the instance is assigned or copied. `Struct` and `union` instances are scalar variables and their entire content is always assigned or copied[1].

A `class` can also contain constraint declarations to control the randomization of data item values; whereas a `struct` and `union` cannot. Finally, it will be possible to modify the default behavior and constraints of a `class` through `inheritance`,

1. Except when passed to a `ref` task or function argument.

without actually modifying the original base model. *`Struct`* and *`union`* do not support inheritance.

Rule 4-54 — *Transactions shall be modeled using transaction descriptors.*

The natural tendency is to model transactions as procedure calls, such as *`read()`* and *`write()`*. This approach makes it more difficult to generate random streams of transactions, constraining transactions and registering transactions with a scoreboard.

Transactions are better modeled using a transaction descriptor. As shown in Example 4-29, a transaction descriptor contains all of the necessary information to ultimately call the appropriate procedure that implements it, which is shown in Example 4-30.

Example 4-29. Transaction Descriptor Object

```
class wb_cycle extends vmm_data;
   ...
   typedef enum {READ, WRITE, ...} cycle_kinds_e;
   rand cycle_kinds_e kind;
   ...
   rand bit [63:0] addr;
   rand bit [63:0] data;
   rand bit [ 7:0] sel;
   ...
   typedef enum {UNKNOWN, ACK, RTY, ERR,
                 TIMEOUT} status_e;
   status_e status;
   ...
endclass: wb_cycle
```

Example 4-30. Transaction Procedures

```
task read(input  bit [63:0] addr,
          output bit [63:0] data,
          input  bit [ 7:0] sel,
          ...);

task write(input logic [63:0] addr,
           input logic [63:0] data,
           input logic [ 7:0] sel,
            ...);
```

Tests never actually call the procedure that implements the transaction. Calling is performed by the transactor itself. Instead, tests submit a transaction descriptor to a transactor for execution. This approach offers several advantages:

1. It is easy to create a series of random transactions. Generating random transactions becomes a process identical to generating random data. Notice how all properties in Example 4-29 have the `rand` attribute.
2. Random transactions can be constrained. Constraints can be applied to object properties only. Constraining the transactions modeled using procedures requires additional procedural code. Procedural constraints, such as weights in a `randcase` statement, cannot be modified without modifying the source code, thus preventing reusability.
3. New transactions can be added without modifying interfaces. A new transaction can be added by simply creating a new variant of the transaction object. No new class is created, no class interface is modified and no testcase is changed.
4. It allows easier integration with the scoreboard. Since a transaction is fully described as an object, a simple reference to that object instance passed to the scoreboard is enough to completely define the stimulus and derive the expected response.

This approach may appear to complicate the writing of directed or manual stimulus. The section titled "Directed Stimulus" on page 219 shows how this type of stimulus can be easily created.

Rule 4-55 — *Data and transaction model classes shall be derived from the* `vmm_data` *class.*

This base class provides a standard set of properties and methods proven to be useful in implementing verification environments and testcases. Furthermore, since SystemVerilog does not have the concept of a *void* or anonymous type, it provides a common base type for writing generic data processing and transfer components. See section titled "vmm_data" on page 383 for more details.

Recommendation 4-56 —*A channel class named* `<class_name>_channel` *should be declared for any class derived from the* `vmm_data` *class.*

The channel object is the primary transaction and data interface mechanism used by transactors. It is implemented using a parametrized class and can be used with any class that is derived from the `vmm_data` class. To simplify the syntax of referring to the channel class type and isolate users from the implementation details of the

channel class, a macro is provided to define the channel class. See section titled "vmm_channel" on page 387 for more details.

Example 4-31. Declaring a Channel Class

```
class wb_cycle extends vmm_data;
   ...
endclass: wb_cycle

‘vmm_channel(wb_cycle)
```

Alternative 4-57 —*Data and transaction descriptor models may be packaged separately.*

Data and transaction models may need to be used by different transactor sets, each implementing or generating the data items or transactions according to different mechanisms or protocols. For example, the model for an Ethernet frame will need to be used by transactors implementing the MAC functionality and transactors implementing the various media-independent physical interfaces, as well as generators that create streams of Ethernet frames.

Example 4-32. Separate Ethernet Frame and Transactor Packaging

In eth_frame.sv:

```
class eth_frame extends vmm_data;
   ...
endclass: eth_frame
...
```

In mii.sv:

```
...
class mii_phy_layer extends vmm_xactor;
   ...
endclass: mii_phy_layer
...
```

Class Properties/Data Members

This section gives directives for properties and methods that can be used to model, transform or operate on data and transactions.

Rule 4-58 — *All data classes shall have a* `public static` *class property referring to an instance of the message service interface.*

This instance of the message service interface is used to issue messages from any data item or transaction instance when a more localized message service interface (such as a transactor) is not readily or clearly available. The class property must be `public` to be controllable.

A class-static instance is used to avoid creating and destroying too many instances of the message service interface as there will be thousands of object instances created and destroyed throughout a simulation.

Example 4-33. Declaring and Initializing a Message Service Interface

```
class eth_frame extends vmm_data;
   static vmm_log log = new("eth_frame", "class");
   ...
   function new();
      super.new(this.log);
   endfunction: new
   ...
endclass: eth_frame
```

Data and transaction descriptors will flow through various transactors in the verification environment. Messages related to a particular data object instance should be issued through the message service interface in the transactor where the need to issue the message is identified. That way, the location of the message source can be easily identified—and controlled. Information about the data or transaction that caused the message can be included in the text of the message or by using the `vmm_data::psdisplay()` method. See Example 4-25 for an example.

Do not provide a new instance of a message service interface with each data or transaction descriptor as this will cause significantly more runtime memory to be used and affect the runtime performance of the message service management procedures. It will not provide more information as the apparent source of the messages will be the same, regardless of the location of the data or transaction descriptor in the verification environment, making it more difficult to localize the problem.

Similarly, do not use the message service interface of the transactor that created the data or transaction descriptor by passing a reference via the data or transaction descriptor constructor. The apparent source of all data-related messages would be that transactor, regardless of its current location in the verification environment.

Rule 4-59 — *All class properties corresponding to a protocol property or field shall have the `rand` attribute.*

The `rand` attribute of a class property can be turned off to make it non-random. However, it cannot be made `rand` after the fact. See Example 4-29 for an example.

Rule 4-60 — *A `rand` class property shall be used to define the kind of transaction being described.*

A class must be able to model all possible kinds of transactions for a particular protocol. Do not use inheritance to describe each individual transaction. Instead, use a class property to identify the kind of transaction described by the instance of the transaction descriptor. See the `kind` class property in Example 4-29 for an example.

Rule 4-61 — *The size of a `rand` array-type class property shall be unconditionally constrained to limit its value.*

If the size of a randomized array is left unconstrained, it may[2] be randomized to an average length of 2^{30}. To avoid this situation, the size of a randomized array should *always* be constrained to a reasonable value. It is a good idea to locate this constraint in a separate constraint block to let it be turned off or overridden.

Example 4-34. Declaring a `class` with a Randomized Array

```
class eth_frame extends vmm_data;
   ...
   rand bit [15:0] len_typ;
   rand bit [ 7:0] data[];
   ...
   constraint valid_len_typ {
      (data.size() <= 1500 && len_typ == data.size())
      || len_typ >= 16'h0600;
   }
   constraint limit_data_size {
      data.size() < 65536;
   }
   ...
endclass: eth_frame
```

2. How arrays are randomized is simulator-specific.

Rule 4-62 — *All class properties with a `rand` attribute shall be public.*

This approach will make it possible to turn off their `rand` attribute, constrain them in a derived class, higher-level classes or via the `randomize-with` statement. If the properties are `local`, none of this will be possible. This rule breaks one of the most basic rules of object-oriented programming. But, these software rules were designed for a programming language where randomization and constraint solving does not exist. See Example 4-29 for an example.

Recommendation 4-63 —*All class properties without a `rand` attribute should be local.*

Object state information should be accessed via public methods. This approach will ensure that the implementation can be modified while preserving the interface. Also, if properties are interrelated, using methods to set their value will ensure they remain consistent.

However, if a set of independent class properties has individual of `set_...()` and `get_...()` methods, these class properties can be made public and the methods eliminated.

Recommendation 4-64 —*Transaction descriptors should have implementation and context references.*

Transaction descriptors for higher-level transactions should have a list of references to the lower-level transactions used to implement them. This list would be added to by lower-level transactors in the verification environment as they implement the higher-layer transaction. The completed list will only be valid when the transaction's processing has ended. A scoreboard can then make use of the list of sub-transactions to determine its status and what response to expect.

Conversely, the descriptor for a low-level transaction should have a reference to the higher-level transaction descriptor it implements. This reference will help the scoreboard or other verification environment components make sense of the transaction and help determine the expected response.

Example 4-35. Transaction Context References

```
class usb_packet extends vmm_data;
   ...
   usb_transaction context_data;
   ...
endclass: usb_packet
```

```
class usb_transaction extends vmm_data;
   ...
   usb_packet packets[];
   usb_transfer context_data;
   ...
endclass: usb_transaction

class usb_transfer extends vmm_data;
   ...
   usb_transaction transactions[];
   vmm_data        context_data;
   ...
endclass: usb_transfer
```

In cases where a transaction may be implemented using different lower-level protocols, the implementation references should be of type *vmm_data* to be able to refer to any transaction descriptor, regardless of the protocol. Similarly, the context reference of a low-level transaction should be of type *vmm_data,* if it can implement or carry information from different higher-level protocols. As shown in Example 4-36, an Ethernet frame can transport any protocol information and thus should have a generic context reference.

Example 4-36. Protocol-Generic Context Reference

```
class eth_frame extends vmm_data;
   ...
   vmm_data context_data;
   ...
endclass: eth_frame
```

Rule 4-65 — *Data protection class properties shall model their validity, not their value.*

The information in CRC, HEC or parity properties is not in the actual value of the class property but in their correctness. These properties must be modeled using a mask, indicating which bit of the class property value is to be valid or corrupted (on transmission) or was valid or not (on reception). A value of zero indicates that the corresponding bit was valid.

The actual value of these properties is computed using methods and inserted or compared in the data object only upon packing and unpacking or physical transmission or reception.

Example 4-37. Frame Check Sequence (FCS) Validity Class Property

```
class eth_frame extends vmm_data;
   ...
   rand bit [31:0] fcs;
   ...
   constraint valid_fcs {
      fcs == 32'h0000_0000;
   }
   ...
   virtual function [31:0] compute_fcs();
      ...
      compute_fcs = this.utils.compute_crc32(...);
   endfunction: compute_fcs
...
endclass: eth_frame
...
function void eth_utils::frame_to_bytes(eth_frame frame,
                                        ...);
      ...
      ... = this.compute_crc32(...) ^ frame.fcs;
endfunction: frame_to_bytes
```

Rule 4-66 — *A constraint block shall keep the value of protection class properties equal to zero by default.*

Protection errors can be injected by overriding or turning the constraint block off. See Example 4-37 for an example.

Rule 4-67 — *Fixed payload data shall be modeled using explicit class properties.*

Some protocols define fixed fields and data in normally user-defined payload for certain data types. For example, fixed-format 802.2 link-layer information may be present at the front of the user data payload in an Ethernet frame. Another example is the management-type frame in 802.11: The content of the user-payload is replaced with protocol management information.

Fixed payload data should be modeled using explicit properties as if they were located in non-user-defined fields. The length of the remaining user-defined portion of the payload should be reduced by the corresponding number of bytes used by the fixed payload data, not modeled in explicit properties.

User-defined data is often modeled as an array of bytes. Leaving it up to the user to correctly interpret or format fixed payload data is error prone. Applying constraints to payload elements becomes cumbersome as fixed data may overlap multiple bytes or be concatenated in the same byte.

Example 4-38. Fixed Payload Format Class Property

```
class eth_frame extends vmm_data;
   ...
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [15:0] opcode;
   rand bit [15:0] pause_time;
   ...
   typedef enum [15:0] {PAUSE = 16'h0001} opcodes_e;
   ...
   constraint valid_pause_frame {
      if (format == CONTROL && opcode == PAUSE) begin
          dst     == 48'h0180C2000001;
          max_len == 42;
      end
   }
   ...
   virtual function int unsigned byte_pack(...);
      ...
      case (format)
      ...
      CONTROL: begin
         ... = 16'h8808;
         ... = this.opcode;
         case (this.opcode)
         PAUSE: begin
            ... = this.pause_time;
         end
         endcase
      end
      ...
   endfunction: byte_pack
...
endclass: eth_frame
```

Rule 4-68 — *Class inheritance shall not be used to model different data formats.*

Data units and transactions often contain information that is optional or is unique to a particular kind of data or transaction. For example, Ethernet frames may or may not have virtual LAN (VLAN), link-layer control (LLC), sub-network access protocol

(SNAP) or control information—in any combination. Another example is the instruction set of a processor where different types of instructions use different numbers and modes of operands.

Using traditional object-oriented design practices, inheritance looks like an obvious implementation: Use a base class for the common properties then extend it for the various differences in format. This approach also seems the natural choice as the SystemVerilog equivalent[3] to ***e***'s *when* inheritance. Using inheritance to model data formats creates three problems, two of which are related to randomization and constraints—concerns that do not exist in traditional object-oriented languages.

The first problem is the difficulty of generating a stream containing a random mix of different data and transactions formats. Because an Ethernet device must be able to accept any mix of various Ethernet frame types on any given port, just like a processor must be able to execute any mix of instructions, it is often necessary to generate a random mix of data items with different formats.

Using a common base class gets around the type-checking problem. However, in SystemVerilog, objects must first be instantiated before they can be randomized. And because instances must be created based on their ultimate type, not their base type, the particular format of a data item or transaction would be determined before randomization of its content. Thus, it would be impossible to use constraints to control the distribution of the various data and transaction formats or to express constraints on a format as a function of the content of some other class property (e.g., if the destination address is equal to this, then the Ethernet frame must have VLAN but no control information).

The second problem is the difficulty of adding constraints to be applied to all formats of a data item or transaction descriptor. To add constraints to a data model, the most flexible mechanism is to create a derived class. To add a constraint that must apply to all formats of a data model cannot be done by simply extending the base class common to all formats as it simply creates yet another class unrelated to the other derivatives. It would require extending each one of the ultimate class extensions.

The final problem is that you will not be able to recombine different but orthogonal format variations. For example, the optional VLAN, LLC and control format variations on an Ethernet frame are orthogonal. This aspect creates eight possible variations of the Ethernet frame. Because SystemVerilog does not support multiple

3. SystemVerilog's inheritance is equivalent to ***e***'s *like* inheritance. *Tagged unions* are the closest constructs to its *when* inheritance.

inheritance, using inheritance to model this simple case will require eight different classes: one for each combination of the presence or absence of the optional information.

This approach creates an exponential number of classes. This problem could be solved if the language supported multiple inheritance—an oft-mentioned grievance against the language—but it would not help in solving the significantly more serious previous two problems. This problem is better solved using proper modeling methodology than a new language capability.

Rule 4-69 — *Composition shall not be used to model different data formats.*

Composition is the use of class instances inside another class to form a more complex data or transaction descriptor. Optional information from different formats can be modeled by instantiating—or not—a class containing that optional information in the data model. If the information is not present, the reference would be set to `null`. If the information is present, the reference would point to an instance containing that information. This technique also suffers from two severe problems with randomization and one minor problem.

The first problem is that the randomization process in SystemVerilog does not allocate sub-instances, even if the reference class property has the `rand` attribute. The randomization process either randomizes a pre-existing instance or does nothing if the reference is `null`.

The second problem is that it complicates the expression of constraints that may involve a `null` reference. A `null` reference would cause a runtime error and constraint guards must be used to detect the absence of the optional properties. Furthermore, it is not possible to express constraints to determine the presence or absence—or their respective ratio—of the sub-instance. It would thus be impossible to define the data format based on some other (possibly random) properties.

The last problem is the needless introduction of hierarchies of references to access properties that, in all respect, belong to the same data or transaction descriptor. One would have to remember whether a class property is optional or not and under which optional instance it is located to know where to access it. However, a runtime error while attempting to access non-existent information in the current data format would be a nice type-checking mechanism. But that benefit does not outweigh the other disavantages.

Rule 4-70 — *`Tagged unions` shall not be used to model different data formats.*

`Unions` allow multiple data formats to coexist in the same bits. `Tagged unions` enforce strong typing in the interpretation of multiple orthogonal data formats. This approach is the SystemVerilog almost-equivalent to ***e***'s `when` inheritance.

Unfortunately, tags cannot be randomized. It is not possible to have a `tagged union` randomly select one of the tags, much less constrain the tag based on other class properties. It is also not possible to constrain fields in randomly-tagged unions because the value of the tag is not yet defined until solved.

Should the restrictions on constraining and randomizing `tagged unions` be eventually lifted, they should be used in lieu of Rule 4-71.

Rule 4-71 — *A class property with the `rand` attribute shall be used to indicate if optional properties from different data formats are present.*

Instead of using inheritance, composition or `tagged unions` to model different data and transaction formats, use the value of a discriminant class property. It will be necessary for methods that deal with the ultimate format of the data or transaction—such as `byte_pack()` and `compare()`—to procedurally check the value of these discriminant properties to determine the format of the data or transaction and decide on a proper course of action.

Example 4-39. Using a Discriminant Class Property to Model Data Format

```
class eth_frame extends vmm_data;
   ...
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand bit [ 2:0] user_priority;
   rand bit        cfi;
   rand bit [11:0] vlan_id;
   ...
   virtual function string
      psdisplay(string prefix = "");
      $sformat(psdisplay,
         "%sdst=48'h%h, src=48'h%h, len/typ=16'h%h\n",
         prefix, da, sa, len_typ);
      case (this.format)
      TAGGED: begin
         $sformat(psdisplay,
```

```
                "%s%s(tagged) cfi=%b pri=%0d, id=12'h%h\n",
                 psdisplay, prefix,
                 cfi, user_priority, vlan_id);
            end
            ...
            endcase
            ...
            $sformat(psdisplay, "%s%sFCS = %0s",
                     psdisplay, prefix,
                     (fcs) ? "BAD" : "good"));
        endfunction: psdisplay
        ...
    endclass: eth_frame
```

Because a single *class* is used to model all formats, constraints can be specified to apply to all variants of a data type. Also, because the format is determined by an explicit class property, constraints can be used to express relationships between the format of the data and the content of other properties. Orthogonal variations are modeled using different discriminant properties allowing all combinations of variations to occur within a single model.

This technique may appear verbose but does not require any more lines of code or statements to fully implement. Inheritance provides for better localization of the various differences in formats, but does not reduce the amount of code—and may even increase it. Furthermore, this technique does not require that the optional properties be modeled in specific locations amongst the other properties to enable some built-in functionality to operate properly. The data and transaction models are implemented to facilitate usage, not match some obscure language or simulator requirement.

However, this approach has an obvious disadvantage: There is no type checking to prevent the access of a class property that is not currently valid given the current value of a discriminant class property.

Suggestion 4-72 — *A discriminant class property may be combined with composition to model different data formats.*

If strong type checking is required, this approach may be combined with composition to create the data or transaction descriptor. The presence or absence of optional class properties is not indicated by a reference to a sub-class that is *null* or not. Instead, the discriminant property indicates that fact. The descriptor can be fully populated before randomization then pruned afterward to eliminate the unused class properties.

However, it may be difficult to ensure the correct construction of a manually-specified descriptor.

Example 4-40. Combining a Discriminant Class Property and Composition

```
class eth_vlan_data;
   rand bit [ 2:0] user_priority;
   rand bit        cfi;
   rand bit [11:0] id;
endclass: eth_vlan_data

class eth_frame extends vmm_data;
   ...
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand eth_vlan_data vlan;
   ...
   function void pre_randomize();
      if (this.vlan == null) this.vlan = new;
   endfunction

   function void post_randomize();
      if (format != TAGGED) this.vlan = null;
   endfunction
   ...
endclass: eth_frame
```

Methods

This section presents guidelines for using methods in data and transaction models.

Recommendation 4-73 —*The constructor should be callable without arguments.*

This style will allow the predefined atomic and scenario generators to be used to generate streams of data or transaction descriptors. If arguments are useful, they should have default values. See Example 4-33 for an example.

Recommendation 4-74 —*All non-local methods should be virtual.*

This structure will allow the functionality of methods to be extended in class derivatives. If a method is not virtual, it will not be possible to modify the behavior of existing code to perform a slightly different task—for example, injecting error—

without modifying the original code. See the specification of the *vmm_data* base class on page 383 for an example.

Rule 4-75 — *All methods shall be functions.*

Methods in data and transaction descriptors should be concerned only with their immediate state. There should not be any need for the simulation time to advance or for the execution thread to be suspended within these methods. See the specification of the *vmm_data* base class on page 383 for an example.

Data and transaction processing requiring time to advance or the execution thread to be suspended should be located in transactors.

Rule 4-76 — *All classes derived from the* `vmm_data` *class shall provide implementations for the* `psdisplay()`, `is_valid()`, `allocate()`, `copy()` *and* `compare()` *virtual methods.*

These methods provide the basic functionality required to implement a verification environment. They have no built-in equivalent in the SystemVerilog language. Refer to “vmm_data” on page 383 for the detailed specification of these methods.

The *vmm_data::allocate()* method is a simple call to *new* and appears redundant. But, it enables the creation of factories and the use of polymorphism in transactors, which is not possible with the direct use of the constructor. Refer to “OOP Primer: Factory Pattern” on page 217 for a short explanation of factories.

The *vmm_data::copy()* method creates a suitable copy of the data or transaction instance. It may be shallow or not; or it may be deep or not. For example, the context references in a descriptor should always be copied shallow. This method hides the details of the class implementation from the user.

Recommendation 4-77 —*All classes derived from the* `vmm_data` *class should provide implementations for the* `byte_size()`, `byte_pack()` *and* `byte_unpack()` *virtual methods.*

It is necessary to implement these methods if a data model needs to be transmitted across a physical interface or between different simulations (e.g., from SystemVerilog to SystemC). Refer to “vmm_data” on page 383 for the detailed specification of these methods.

SystemVerilog does not define *packed* classes. Yet, in many instances, a data item must be transmitted over a certain number of byte lanes across a physical interface.

The same stream of data, received over the physical interface must be interpreted back into higher level structure and information. This is automatically handled by *packed struct* and *unions*, but not in classes. The advantages and flexibility offered by *classes* is not worth sacrificing for this simple built-in operation in other data structures. The same functionality can be encapsulated in those predefined methods.

The implementation of the *byte_pack()* method shall only pack the relevant properties based on the value of discriminant properties. Not all properties may be valid or relevant under all possible data or transaction formats. The packing methods must check the value of discriminant properties to determine which class property to include in the packed data, in addition to their format and ordering. See Example 4-38 for an example.

Rule 4-78 — *The vmm_data::byte_unpack() method shall interpret the packed data and set discriminant properties appropriately.*

Often, discriminant properties are logical properties, not directly packed into bit-level data nor directly unpacked from it. However, the information necessary to identify a particular variance of a data object is usually present in the packed data. For example, the value *16'h8100* in bytes 12 and 13 of an Ethernet MAC frame stream indicate that the VLAN identification fields are present in the next two bytes. If the information about the data format is not available in the bytes to be unpacked, the optional *kind* argument may be used to specify a particular format to expect.

The unpacking methods must interpret the packed data and set the value of the discriminant properties accordingly. Similarly, it must set all relevant properties to their interpreted values based on the interpretation of the packed data. Properties not present in the data stream should be set to unknown or undefined values.

Example 4-41. Unpacking an Ethernet Frame

```
class eth_frame extends vmm_data;
   ...
   typedef enum {UNTAGGED, TAGGED, CONTROL}
      frame_formats_e;
   rand frame_formats_e format;
   ...
   rand bit [47:0] dst;
   rand bit [47:0] src;
   rand bit        cfi;
   rand bit [ 2:0] user_priority;
   rand bit [11:0] vlan_id;
   ...
   virtual function int unsigned byte_unpack(
```

```
            const ref logic [7:0] array[],
            input int unsigned    offset = 0,
            input int             len    = -1,
            input int             kind   = -1);
            integer i;

            i = offset;
            this.format = UNTAGGED;
            ...
            if ({array[i], array[i+1]} === 16'h8100) begin
               this.format = TAGGED;
               i += 2;
               ...
               {this.user_priority, this.cfi, this.vlan_id} =
                  {array[i], array[i+2]};
               i += 2;
               ...
            end
            ...
         endfunction: byte_unpack
         ...
      endclass: eth_frame
```

Rule 4-79 — *A virtual method shall be provided to compute the correct value of each data protection class property.*

Because the data protection class property is encoded simply as being valid or not, it must be possible to derive its actual value by other means when necessary. The method must be virtual to allow for the introduction of a different protection value computation algorithm if necessary. The packing method is responsible for corrupting the value of a data protection class property if it is modeled as invalid, not the computation method. See Example 4-37 for an example.

Constraints

Rule 4-80 — *A constraint block shall be provided to ensure the validity of randomized class property values.*

Some properties may be modeled using a type that can yield invalid values. For example, a *`length`* class property may need to be equal to the number of bytes in a payload array. This constraint would ensure that the value of the class property and the size of the array are consistent. Note that "valid" is not the same thing as "error-free." Validity is a requirement of the descriptor implementation, not of the data or transaction being described.

This constraint block must never be turned off nor overridden; hence, it is a good idea to use a unique name, such as "*class_name_valid*".

Example 4-42. Basic Frame Validity Constraint Block

```
class eth_frame extends vmm_data;
   ...
   rand int unsigned min_len;
   rand int unsigned max_len;
   ...
   constraint eth_frame_valid {
      min_len <= max_len;
   }
   ...
endclass: eth_frame
```

Recommendation 4-81 —*Constraint blocks should be provided to produce better distributions on size or duration class properties.*

Size and duration properties do not have equally interesting values. For example, short or back-to-back and long or drawn-out transactions are more interesting than average transactions. Randomized class properties modeling size, length, duration or intervals should have a constraint block that distributes their value equally between limit and average values.

Example 4-43. Constraint Block to Improve Distribution

```
class eth_frame extends vmm_data;
   ...
   constraint interesting_data_size {
      data.size() dist {min_len               :/ 1,
                        [min_len+1:max_len-1] :/ 1,
                        max_len               :/ 1};
   }
   ...
endclass: eth_frame
```

A similar modification of value distributions should be implemented to increase the likelihood that corner cases will be generated. However, the definition of a corner-case is usually DUT-specific. Any constraint designed to hit DUT-specific corner cases must be implemented in a class extension of the data or transaction descriptor,

not in the descriptor class itself. This implementation will avoid polluting a reusable data or transaction model with DUT-specific information.

Example 4-44. Adding DUT-Specific Corner-Case Constraints

```
class long_eth_frame extends eth_frame;
   constraint long_frames {
      data.size() == max_len;
   }
   ...
endclass: long_eth_frame
```

Rule 4-82 — *A distribution constraint block shall constrain a single class property.*

Use one constraint block per class property to make it easy to turn off or override without affecting the distribution of other properties. See Example 4-43 for an example.

Recommendation 4-83 —*Discriminant class properties should be solved before dependent class properties.*

A conditional constraint block does not imply that the properties used in the expression are solved before the properties in the body of the condition. If a class property in the body of the condition is solved with a value that implies that the condition cannot be true, this result will further constrain the value of the properties in the condition. If there is a greater probability of falsifying the condition, it is less likely to get an even distribution over all discriminant values.

In Example 4-45, if the *length* class property is solved before the *kind* class property, it is unlikely to produce *CONTROL* packets because there is a low probability of the *length* class property to be solved as 1.

Example 4-45. Poor Distribution Caused by Conditional Constraints

```
class some_packet;
   typedef enum {DATA, CONTROL} kind_typ;
   rand kind_typ kind;

   rand int unsigned length;
   ...
   constraint valid_length {
      if (kind == CONTROL) length == 1;
   }
endclass: some_packet
```

This problem can be avoided and a better distribution of discriminant properties can be obtained by forcing the solving of the discriminant class property before any dependent class property using the `solve before` constraint.

Example 4-46. Improved Distribution Caused by Conditional Constraints

```
class some_packet;
   typedef enum {DATA, CONTROL} kind_typ;
   rand kind_typ kind;

   rand int unsigned length;
   ...
   constraint valid_length {
      if (kind == CONTROL) length == 1;
      solve kind before length;
   }
endclass: some_packet
```

Rule 4-84 — *Constraint blocks shall be provided to avoid errors in randomized values.*

Error can be randomly injected by selecting an invalid value for error protection properties. A constraint block should keep the value of such properties valid by default. See Example 4-37 for an example.

Rule 4-85 — *An error-prevention constraint block shall constrain a single class property.*

Use one constraint block per error injection class property to make it easy to turn off or override without affecting the correctness of other properties.

Recommendation 4-86 —*Undefined external* `constraint` *blocks named "test_constraintsX" should be declared.*

External `constraint` blocks are defined outside of the `class` that declares them. If left undefined, they are considered empty and do not add any constraints to the `class` instances. These `constraint` blocks can be defined later by individual tests to add constraints to all instances of the `class`. See Alternative 5-21 on page 229.

Example 4-47. Declaring Undefined External *constraint* Blocks

```
class eth_frame extends vmm_data;
   ...
   extern constraint test_constraints1;
   extern constraint test_constraints2;
   extern constraint test_constraints3;
   ...
endclass: eth_frame
```

TRANSACTORS

The term *transactor* is used to identify components of the verification environment that interface between two levels of abstractions for a particular protocol or to generate protocol transactions. In Figure 4-2, the boxes labelled Driver, Monitor, Checker and Generator are all transactors. The lifetime of transactors is static to the verification environment: They are created at the beginning of the simulation and stay in existence for the entire duration[4]. They are structural components of the verification components; they are similar to *modules* in the DUT. Only a handful of transactors get created. In comparison, transactions have a dynamic lifetime: Thousands get created by generators, flow through transactors, get recorded and compared in scoreboards then freed.

Traditional bus-functional models are called *command-layer* transactors. Command-layer transactors have a transaction-level interface on one side and a physical-level interface on the other. Functional-layer and scenario-layer transactors only have transaction interfaces and do not directly interface to physical signals.

This section specifies guidelines designed to implement transactors that are reusable, controllable and extendable. Note that reusability, controllability and extensibility are not goals in and of themselves. These features will enable transactors to be reused by different testcases and different verification environments. They will enable transactors to be controlled to meet the specific needs of specific testcases, and they can be extended to include the features required by specific environments. This control and extension must be accomplished without modifying the transactors themselves to avoid compromising the correctness of known-good transactors or to modify the behavior or functionality of existing testcases.

4. Some classes of reconfigurable designs may require dynamically reconfigurable verification environment topologies. In that case, transactors will have a more dynamic lifetime.

Rule 4-87 — *All transactor-related declarations shall have a unique prefix.*

Transactors will need to be used by different verification environments. Different environments will require different combinations of transactors. Using a unique prefix for all global name-space declarations will prevent collisions with other transactors.

Example 4-48. MII Transactors

```
class mii_cfg;
   ...
endclass: mii_cfg
...
class mii_mac_layer extends vmm_xactor;
   ...
endclass: mii_mac_layer
...
class mii_phy_layer extends vmm_xactor;
   ...
endclass: mii_phy
```

Rule 4-88 — *All transactor-related declarations shall be in the same file.*

All declarations required by a transactor must be packaged together. Using a single file to package all these related declarations simplifies the task of bringing all necessary declarations required to use a transactor into a simulation. This rule is consistent with Rule 4-4.

Rule 4-89 — *Transactors shall be usable in both the active and reactive regions.*

Using transactors in the reactive region will ensure that they avoid race conditions between the design, the assertions and the verification environment, as required by Rule 4-22. When using transactors in transaction-level models, they should execute in the active region, similar to a RTL model, to avoid the same DUT-testbench race conditions.

To be usable in both the active and reactive region, transactors must be declared in a scope that is accessible by both *`modules`* and *`program`* blocks.

Example 4-49. Transactors Declared in *`$root`*

```
class mii_mac_layer extends vmm_xactor;
   ...
endclass: mii_mac
```

Suggestion 4-90 — *Transactor-related declarations may be packaged in a* `package`.

Using a `package` to package all related declarations may offer the opportunity for separate compilation in some tools. Although a `package` appears to eliminate the need for a unique prefix, the potential to use the "`import pkgname::*`" statement still necessitates the clear differentiation of names that may potentially clash in the global name space.

Example 4-50. MII Transactor Package

```
package mii;

class mii_cfg extends vmm_data;
   ...
endclass: mii_cfg
...
class mii_mac_layer extends vmm_xactor;
   ...
endclass: mii_mac_layer
...
class mii_phy_layer extends vmm_xactor;
   ...
endclass: mii_phy_layer
...
endpackage: mii
```

Rule 4-91 — *Transactors shall be implemented using a* `class`.

Transactors and data are both implemented using the `class` construct. The difference between a transactor `class` and a data `class` is their lifetime. Transactor instances are created in limited number at the beginning of the simulation and remain in existence through out. Data and transaction descriptors instances are created in very large number throughout the simulation and have a short life span. In that respect, transactor `classes` are used much like `modules`: modules instances, too, are static throughout the simulation. The current state of each transactor is maintained in local properties, and the execution threads are implemented in local methods.

`Classes` are used instead of `modules` because their instantiation is performed at run time. Therefore, the structure of the verification environment can be dynamically configured[5] according to the generated testcase configuration descriptor. Modules,

5. And their topology dynamically reconfigured if necessary.

being instantiated during the elaboration phase, define a structure before the simulator has had the chance to randomize the testcase configuration descriptor.

Classes are also preferred because they offer an implementation protection mechanism. It is possible to limit the access to various properties and methods in the class by declaring them as *protected* or *local*. No such protection mechanism exists in *modules*. Protecting the implementation of a *class* lets the implementation control the interface that is exposed to the user, and this protection lets the implementation be modified in a backward-compatible fashion. *Modules*, with their unrestricted access to all of their internal constructs, may put the implementor in a straitjacket if users use internal state information and procedures.

Classes also offer the opportunity to provide basic shared functionality to all transactors through a shared base class. Because *modules* are not built on the object-oriented framework, they cannot be used to offer such shared functionality.

Rule 4-92 — *Transactors shall be implemented in classes derived from* `vmm_xactor`.

The *vmm_xactor* base class contains standard properties and methods to configure and control transactors. To ensure that all transactors have a consistent usage model, they must be derived from a common base class. Refer to “vmm_xactor” on page 411 for the detailed specification of this class.

Rule 4-93 — *All threads shall be started in the extension of the* `vmm_xactor::main()` *task.*

The runtime behavior of transactors is controlled by the *vmm_xactor::start_xactor()* and *vmm_xactor::reset_xactor()* functions. For these functions to work properly, all threads that implement autonomous behavior for a transactor must be forked in the body of the *vmm_xactor::main()* task.

Rule 4-94 — *No threads shall be started in the constructor.*

This rule is a corollary of the previous guideline. Threads started in the constructor cannot be controlled by the *vmm_xactor::start_xactor()* and *vmm_xactor::reset_xactor()* methods.

It is important that no threads be started as soon as a transactor is instantiated. When the verification environment is initially built and the transactor instantiated, the DUT may not be yet ready to receive stimulus. Transactors and generators may have to be

suspended until the DUT has been properly configured. Furthermore, if a testcase needs to inject directed stimulus, it must be able to suspend a transactor or generator for the entire duration of the simulation. If that transactor or generator has already had the opportunity to generate stimulus, it may be impossible to write the required directed testcase.

Rule 4-95 — *Extensions of the `vmm_xactor::main()` task shall call `super.main()`.*

Transactors may be implemented as successive derived classes all based on the `vmm_xactor` class. Each inheritance layer may include relevant autonomous threads started in their extension of their respective `vmm_xactor::main()` task. The execution of the implementation of this task in all intermediate extensions of the `vmm_xactor` base class is necessary for the proper operation of the transactor and control methods.

Example 4-51. Extension of the `vmm_xactor::main()` Task

```
task mii_mac_layer::main();
   super.main();
   ...
endtask: main
```

Rule 4-96 — *The `vmm_xactor::start_xactor()`, `vmm_xactor::stop_xactor()` and `vmm_xactor::reset_xactor()` functions shall be extended to add protocol or transactor-specific functionality.*

These methods are virtual to enable the addition of functionality specific to the implementation of a transactor or a protocol to be executed when a transactor is started, stopped or reset. Should a transactor or protocol not have specific functionality to be executed at these control points, the functions should still be extended to allow further extension of the transactor class to overload these virtual methods if necessary. Refer to “vmm_xactor” on page 411 for the detailed specification of these methods.

Rule 4-97 — *Extensions of the `vmm_xactor::start_xactor()`, `vmm_xactor::stop_xactor()` and `vmm_xactor::reset_xactor()` shall call their implementation in the base class using the `super` prefix.*

The implementation of a virtual method in a base class that has been overloaded in a derived class is only invoked when implicitly called using the `super` prefix. When a

transactor extends these methods to perform transactor or protocol-specific operations, they must invoke the implementation of these virtual methods in the base class for proper operation.

Example 4-52. Extension of a Control Method

```
function void
   mii_mac_layer::reset_xactor(reset_e typ = SOFT_RST);
   super.start_xactor(typ);
   ...
endfunction: reset_xactor
```

Rule 4-98 — *Layers of a protocol shall be modeled as separate transactors.*

Protocols are often specified using a layering concept, each with different levels of abstraction. The transactors implementing these protocols should follow a similar division. The functional layer of the verification environment is built using sub-layers of relevant transactors. For example, a USB functional layer could be composed of USB transaction (host or endpoint) and USB transfer (host controller or device) sub-layers.

Rule 4-99 — *Transactors shall be identified—or configurable—as proactive, reactive or passive.*

Proactive transactors initiate transactions. Reactive transactors respond to transactions. A passive transactor will simply observe the interface in both directions, reporting observed data as it flows by and any protocol violation it observes. The verification environment must be able to control the timing of transactions initiated by proactive transactors, but the verification environment has no control over the initiation or type of transactions observed by reactive or passive transactors.

When modeling reactive and passive transactors, care must be taken so that no data is lost if the transactor is executing user-defined callbacks while a significant event occurs on the upstream interface.

This guideline does not imply that a transactor shall be dynamically reconfigurable, for example from proactive to passive. Due to the significant differences in behavior between modes, it is acceptable to provide this optional configurability at construction-time.

Recommendation 4-100 —*For every proactive or reactive transactor, there should be a passive transactor.*

Proactive and reactive transactors are used when direct interaction with an interface is required to complete or initiate a transaction. When the DUT is embedded into a system, that interface may no longer be controllable and instead, is controlled by another block in the system.

A passive transactor should be available to monitor the transactions that used to be under the control of the block-level environment to be able to reuse the block-level functional coverage model or self-checking structure.

Rule 4-101 —*All messages issued by a transactor instance shall use the message service interface in the* `vmm_xactor::log` *class property.*

This usage will ensure that all messages from that transactor have a consistent format and can be controlled as a single set of messages.

Recommendation 4-102 —*Transactor objects should indicate the occurrence of significant protocol and execution events via the notification service interface in the* `vmm_xactor::notify` *class property.*

These notifications can be used by the verification environment to synchronize with the occurrence of a significant event in a transactor or a protocol interface. When relevant, status information about the reason of the event occurrence should be supplied by the transactor and attached to the notification.

Rule 4-103 —*Transactors shall assign the value of their* `vmm_xactor::stream_id` *class property to the* `vmm_data::stream_id` *class property of the data and transaction descriptors flowing through them.*

The stream identifiers used to set the stream identifier in data and transaction descriptors as they flow through should be set by the transactor and reported to user-defined code extensions in callback methods. This identifier may be used to differentiate between multiple instantiations of the same transactor or the path taken by a data item or transaction descriptor through a verification environment.

Rule 4-104 — *Transactors shall be configurable if the protocol they implement has options.*

Even though designs may be implemented using the same interface protocols, there may be differences in how the protocol is physically implemented by different designs. Optional elements of the protocol, such as bus width, the number of outstanding transactions, clock frequency or the presence of optional side-band signals shall be configurable.

Rule 4-105 — *Transactors shall be configured using a randomizable configuration descriptor.*

The configuration of a transactor must be specified using a configuration descriptor. All the properties in the configuration descriptor must have the `rand` attribute to allow the generation of a random configurations—both to verify the transactor itself under different conditions and to make it usable as a component of the testcase configuration descriptor.

Example 4-53. MII Transactor Configuration Descriptor

```
class mii_cfg;
   rand bit is_100Mb;
   rand bit full_duplex;
endclass: mii_cfg
```

Rule 4-106 — *Transactor configuration descriptor shall be passed via the constructor.*

A transactor must be configured before being used. The best way to ensure that the transactor is configured is to require the configuration descriptor be provided as a constructor argument. The transactor may choose to keep a reference to the original configuration descriptor instance or make a copy of it.

Recommendation 4-107 — *A `reconfigure()` method accepting a new configuration descriptor should be provided to dynamically reconfigure a transactor.*

It should be possible to modify the configuration of a transactor during the simulation. Merely modifying the original configuration descriptor instance may not be sufficient as the transactor has no means of efficiently detecting such a change, or it may have an internal copy different from the original instance. Calling a "*reconfiguration*" method ensures that the transactor is properly notified of the need for a different configuration.

Reconfiguring a transactor that is running may yield unexpected results. The *`reconfigure()`* method may invoke the *`vmm_xactor::reset_xactor()`* function.

Physical-Level Interfaces

Command-level transactors and bus-functional models are components of the command layer. They translate transaction requests from the higher layers of the verification environment to physical-level signals of the DUT. In the opposite direction, they monitor the physical signals from the DUT or between two DUT modules. They also notify the higher layers of the verification environment of any transactions initiated by the DUT.

The physical-level interface of command-layer transactors must interact with the signal-layer construct. As such, they must follow the guidelines outlined in section “Signal Layer” on page 107.

Rule 4-108 — *Physical interfaces shall be specified using a `virtual modport interface` as an argument to the transactor constructor.*

This specification lets each instance of a transactor be connected to a specific interface instance without hardcoding a signal naming or interfacing mechanism. The signal layer creates the necessary *`interface`* instances in the top-level module. The appropriate *`interface`* instance is specified when constructing a transactor to connect that transactor to that *`interface`* instance.

Example 4-54. Virtual Interface in Constructor

```
interface mii_if;
   ...
   modport mac_layer(...);
   ...
endinterface: mii_if
...
class mii_mac_layer extends vmm_xactor;
   ...
   function new(...,
                virtual mii_if.mac_layer sigs,
                ...);
      ...
   endfunction: new
   ...
endclass: mii_mac_layer
```

Rule 4-109 — *The `virtual interface` shall be stored in a public class property.*

This structure will let testcases access the physical interface signals, if required.

Example 4-55. Virtual Interface in Properties

```
class mii_mac_layer extends vmm_xactor;
   virtual mii_if.mac_layer sigs;
   ...
   function new(...,
                virtual mii_if.mac_layer sigs,
                ...);
      ...
      this.sigs = sigs;
      ...
   endfunction: new
   ...
endclass: mii_mac_layer
```

Rule 4-110 — *Command-layer transactors shall not refer directly to clock signals.*

The `clocking` block separates timing and synchronization of synchronous signals from the reference signal. It defines the timing and sampling relationships between synchronous data and clock signals. If a transactor waits for the next edge of the clock by using an `@(posedge...)` statement, it may wait for the wrong active edge—or the wrong clock signal—compared to the one specified in the `clocking` block and sample the wrong value of the synchronous signals. To wait for the next cycle of synchronous signals, use the `@` operator with a clocking block reference.

Example 4-56. Waiting for the Next Clock Cycle

```
task mii_mac_layer::tx_driver();
   ...
   @this.sigs.mtx;
   this.sigs.mtx.txd <= nibble;
   ...
endtask: tx_driver

task mii_mac_layer::rx_monitor();
   ...
   @(this.sigs.mrx);
   if (this.sigs.mrx.rx_dv !== 1'b1) break;
   a_byte[7:4] = this.sigs.mrx.rxd;
   ...
endtask: rx_monitor
```

TRANSACTION-LEVEL INTERFACES

This section applies to the transaction-level interfaces of transactors. In command-layer transactors—drivers and monitors, the transaction-level interface lets the higher layers of the verification environment stimulate the DUT by specifying which transactions should be executed or be notified of which transactions have been observed at a given point in time on a DUT interface.

Transaction-level interfaces remove the higher-level layers from the physical interface details. In functional-layer transactors, a high-level transaction interface is used to receive a description of high-level transactions to be executed. These high-level transactions are executed—or implemented—by executing one or more lower-level transactions, generally of a different type. These lower-level transactions are submitted to a lower level transactor—in a lower functional sub-layer or in the command layer via a low-level transaction interface. For example, an IP packet transaction could be segmented into one or more IP segments by a segmented transactor. Transaction-level interfaces are mechanisms to exchange transactions between two transactors or a directed testcase and a transactor.

Because transactions are specified using descriptors, a conduit can be used to exchange these descriptors between two transactors. A connection between two transactors or a testcase and a transactor is established by having each endpoint refer to the same conduit, as illustrated in Figure 4-7. The connection can be made by instantiating the endpoints in any order to allow the building of verification environments in a bottom-up or top-down fashion. The conduit allows a transactor—whether upstream or downstream—to be connected to any other transactor with a compatible conduit, without requiring any source code modifications.

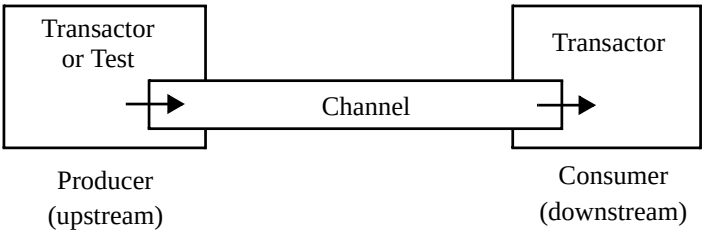


Figure 4-7. Transaction Interface Channel

Traditionally, transaction-level interfaces have been implemented using procedure calls in the transactors themselves. But invoking a procedure in a transactor instance requires having a reference to that transactor in the first place. This requires that verification environments be built in a bottom-up fashion, with the higher-layers

having a reference to the lower-level transactor instances so they can call methods in them. This structure creates some difficulties:

1. You cannot build a verification environment on top of the physical layer that can then be retargeted, without modifications, to a different physical-layer implementation.
2. You cannot build a verification environment on a transaction-level model that can be reused, unmodified, on an implementation with physical-level interfaces.

By encapsulating the transaction exchange mechanism into a conduit, the transactors are considered as endpoints that can be replaced easily, without any knowledge required by or of the other endpoint.

Rule 4-111 — *A channel shall be used to exchange transactions between two transactors.*

Each connection between two endpoints requires a channel instance. An instance cannot be shared by more than one connection.

The `vmm_channel` is a template class that must be specialized based on the data or transaction descriptor it will carry. Based on a previous guideline, a channel class name should have already been defined for every `vmm_data` derivative as the data or transaction descriptor class name with the "`_channel`" suffix. For example, the class `eth_frame_channel` should already exist to carry instances of the `eth_frame` class. Refer to "vmm_channel" on page 387 for the detailed specification of this class.

An `interface` cannot be used as a transaction-level interface because, like a `module`, it is a static construct. It would not be possible to create dynamically reconfigurable verification environments. Furthermore, `interfaces` are not built on top of the object-oriented framework and cannot be derived from one another. It would thus not be possible to provide common functionality through a base `interface` like it is possible through a channel base `class`.

Rule 4-112 — *References to channel instances shall be stored in public class properties suffixed with "`_chan`".*

This structure lets the connection between two transactors be made in arbitrary order. The first one creates the channel instance, then the second one uses the reference to the channel in the first one.

Furthermore, this structure lets directed portions of tests put manually created transaction descriptors into a channel by suspending the execution of the upstream transactor and accessing the channel's *put()* method.

Example 4-57. Channel Reference Properties

```
class mii_mac_layer extends vmm_xactor;
   eth_frame_channel tx_chan;
   eth_frame_channel rx_chan;
   ...
endclass: mii_mac_layer
```

Recommendation 4-113 —*Channel instances should be specified as optional constructor arguments.*

Connecting two transactors requires that they be endpoints on the same channel instance. It should be possible to specify channel instances to connect to as optional constructor arguments. If none are specified, new instances are internally allocated.

Example 4-58. Top-Down Connection of Two Transactors

```
eth_mac      mac = new(...);
mii_mac_layer mii = new(..., mac.pls_tx_chan, ...);
```

Example 4-59. Bottom-Up Connection of Two Transactors

```
mii_mac_layer mii = new(...);
eth_mac        mac = new(..., mii.tx_chan, ...);
```

Example 4-60. Third-Party Connection of Two Transactors

```
eth_frame_channel tx_chan = new(...);
eth_mac      mac = new(..., tx_chan, ...);
mii_mac_layer mii = new(..., tx_chan, ...);
```

This connection requires that channel instances be allocated if none are specified via the constructor argument list.

Example 4-61. Optionally Specifying Channel Instances in Constructor

```
class mii_mac_layer extends vmm_xactor;
   eth_frame_channel tx_chan;
   eth_frame_channel rx_chan;
   ...
   function new(...
                eth_frame_channel tx_chan = null,
                eth_frame_channel rx_chan = null, ...);
      ...
      if (tx_chan == null) tx_chan = new(...);
```

```
            this.tx_chan = tx_chan;
            if (rx_chan == null) rx_chan = new(...);
            this.rx_chan = rx_chan;
        endfunction: new
        ...
    endclass: mii_mac_layer
```

Rule 4-114 — *A transactor shall not hold an internal reference to a channel instance while it is stopped or reset.*

If a transactor holds a copy of the reference to a channel instance in an internal variable, the channel instance cannot be substituted with another one to modify the output or input of a transactor and dynamically reconfigure the structure of a verification environment. While unavoidable during normal operations, a reset or stopped transactor should "release" all such internal references to let the channel instance be replaced.

Rule 4-115 — *Reactive and passive transactors shall allocate a new transaction descriptor instance from a factory instance using the* `vmm_data::allocate()` *method.*

It is often desirable to add user-defined environment-specific or testcase-specific information to a transaction descriptor. This addition must be done via class inheritance, not by modifying the original class, to avoid proliferating unrelated application-specific information into a generic definition—thus lowering its reusability. The problem is that these additional properties are located in a different type from the original type.

This problem manifests itself in reactive and passive transactors that monitor and report transactions observed on a physical or lower-level transaction interface. Transaction descriptor instances are created internally when a new transaction is detected. Because these transactors are written in terms of the original base class, they will allocate an instance of the original, generic class without the required additional information if a call to `new` is used. This problem can be solved by creating the new instances by copying from a factory instance. The `vmm_data::allocate()` and `vmm_data::copy()` methods, being virtual, will allocate an instance of the derived class found in the factory instance, not the original class.

Example 4-62. Reusable Reactive or Passive Transactor

```
class mii_phy_layer extends vmm_xactor;
   ...
   eth_frame_channel rx_chan;
   ...
```

```
      local eth_frame rx_factory;
      ...
      function new(..., eth_frame rx_factory = null);
         ...
         if (rx_factory == null) rx_factory = new;
         this.rx_factory = rx_factory;
         ...
      endfunction: new
      ...
      virtual task rx_monitor();
         ...
         forever begin
            eth_frame fr;
            ...
            $cast(fr, this.rx_factory.copy());
            ...
            this.rx_chan.sneak(fr);
         end
      endtask: rx_monitor
      ...
   endclass: mii_phy_layer
```

Example 4-63. Adding User-Defined Transaction Information

```
class annotated_eth_frame extends eth_frame;
   ...
endclass: my_eth_frame
...
class tb_env extends vmm_env;
   virtual function void build();
      annotated_eth_frame ann_fr;
      ...
      ann_fr = new;
      ...
      this.phy = new(..., ann_fr);
   end
endclass: tb_env
```

Rule 4-116 — *A transactor shall not be both a producer and a consumer for a channel instance.*

Channels cannot enforce which transactor endpoint is the producer and which one is the consumer. Transaction descriptors "flow" from the *`vmm_channel::put()`* method to the *`vmm_channel::get()`* method. A producer is defined by the simple fact that it calls the *`put()`* method, whereas a consumer is defined by the fact that it calls the *`get()`* method. A transactor cannot be both a consumer and producer for the same channel, unless the channel is used internally and not as a transaction interface.

If a bidirectional interface is required, two channel instances must be used, one for each direction.

A functional layer monitor that can be configured as reactive or passive appears to break this rule. When in reactive mode, the response channel (see section titled "Reactive Response" on page 192) is an output channel. When in passive mode, the response channel is an input channel. The monitor will be a producer when configured as reactive but a consumer when configured as passive, but never both at the same time. This guideline is therefore followed.

Rule 4-117 — *Reactive or passive transactors shall use the* `vmm_channel::sneak()` *method to put transaction descriptors in their output channels.*

A reactive or passive transactor may block its execution thread on the execution of the `vmm_channel::put()` method if the channel is full. This blocking action may break the implementation of the protocol and cause data to be missed or checks not to be performed. A transactor should not be written to rely on another endpoint to constantly drain an output channel. Using the `vmm_channel::sneak()` function cannot block, even if the channel is full, and prevents these problems from occurring.

Because the rate of execution of reactive and passive transactors is regulated by the interface (physical or transaction-level) they are monitoring, using the `vmm_channel::sneak()` function should not cause an infinite execution loop in the monitoring thread.

Completion and Response Models

Transaction descriptors are provided to and reported by transactors via a channel instance. It is usually important for the higher-layer transactors to know when a transaction has been completed or how to respond to a reactive transactor. Furthermore, it must be possible for a proactive or reactive transactor to output status information about the execution of the transaction.

The following guidelines will help choose a completion or response model suitable for the transactor and protocol being implemented. A completion model is used by proactive and reactive transactors to indicate the end of a transaction execution. A response model is used by a reactive transactor to request, from the higher layers of a verification environment, additional data or information required to complete a suitable response to the transaction being reacted to.

The completion and response models are described using a producer transactor and a consumer transactor. The consumer transactor executed transactions as requested by the producer transactor and indicates completion and response information back to the producer transactor. The completion model is the same for stimulus and monitor transactors. If the transactor pair was in a stimulus stack, the execution flow would be toward the DUT. If the transactor pair was in a monitoring or reactive stack, the execution flow would be away from the DUT.

Rule 4-118 — *Transactors shall clearly document the completion model used by input channels.*

The completion model used by a transactor to indicate the completion of a transaction is crucial to its proper usage. Each transactor must document the completion model used.

Rule 4-119 — *Reactive transactors shall clearly document the response model expected by output channels.*

The response model used by a reactive transactor to request additional response information to complete a transaction is crucial to its proper usage. Each reactive transactors must document the expected response model.

In-Order Atomic Execution Model

Transactors with an in-order atomic execution model perform transactions in the same order as they were submitted. Each transaction is executed only once and completes in a single execution attempt. Such transactors use a *blocking* completion model. As illustrated in Figure 4-8, the execution thread from the producer transactor (depicted as a dotted line) is blocked while the transaction flows through the channel and is executed by the consumer transactor. It remains blocked until the execution of the transaction is completed.

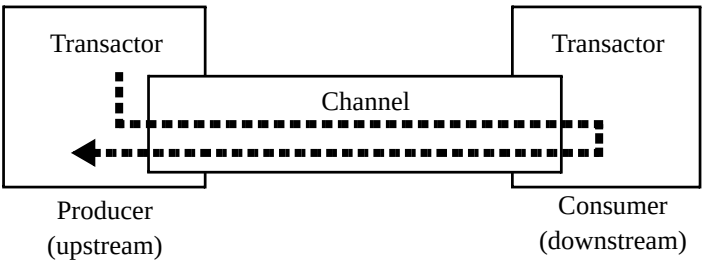


Figure 4-8. In-Order Atomic Completion Model

From the producer transactor's perspective, the blocking completion model is embodied in the *vmm_channel::put()* method. When this method returns, the transaction is completed. Additional completion status information may have been added to the transaction descriptor by the consumer transactor.

Example 4-64. Upstream of a Blocking Completion Model

```
class producer extends vmm_xactor;
   ...
   virtual task main();
      ...
      ... begin
         transaction tr;
         ...
         do
            out_chan.put(tr);
         while (tr.status == RETRY);
         ...
      end
      ...
   endtask: main
endclass: producer
```

The suitability and proper implementation of this completion model requires adhesion to the following guidelines by the consumer transactor:

Rule 4-120 — *Input channel instances shall be reconfigured with a full level of one.*

It is the channel instance that will block the execution of the *vmm_channel::put()* method, not the consumer transactor. That can only happen if the channel is considered full as soon as a transaction is put into the channel itself. Any other configuration would create a *non-blocking* interface.

To ensure that input channels have a full level of one, consumer transactors must explicitly reconfigure the input channel instances. Otherwise, externally created instances with incompatible configurations may be used.

Example 4-65. Reconfiguring an Input Channel Instance

```
class consumer extends vmm_xactor;
   transaction_channel in_chan;
   ...
   function new(transaction_channel in_chan = null);
      ...
      if (in_chan == null) in_chan = new(...);
      in_chan.reconfigure(1);
      this.in_chan = in_chan;
```

```
      endfunction: new
      ...
endclass: consumer
```

Rule 4-121 — *Transaction descriptors shall be peeked from the input channel.*

To keep the *vmm_channel::put()* method blocked for the producer transactor, the channel must not be emptied while the transaction is being executed. Therefore, the *vmm_channel::peek()* or *vmm_channel::activate()* method must be used to obtain the next transaction to be executed from the input channel.

Example 4-66. Peeking Transaction Descriptors

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         this.in_chan.peek(tr);
         ...
         this.in_chan.get(tr);
      end
   endtask: main
   ...
endclass: consumer
```

Rule 4-122 — *Transaction descriptors shall be removed from the channel only when the transaction execution is completed.*

This rule is a corollary to the previous guideline. A transaction is removed from a channel by using the *vmm_channel::get()* or *vmm_channel::remove()* method.

Recommendation 4-123 — *The vmm_data::STARTED and vmm_data::ENDED notifications should be indicated.*

An producer transactor may choose to use a non-blocking model by forking the thread that puts the transaction descriptor into the input channel. Providing a built-in indication of the execution of the transaction will eliminate the need for additional synchronization infrastructure in the producer transactor.

Example 4-67. Indicating Transaction Execution Notifications

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         this.in_chan.peek(tr);
         tr.notify.indicate(vmm_data::STARTED);
         ...
         tr.notify.indicate(vmm_data::ENDED, ...);
         this.in_chan.get(tr);
      end
   endtask: main
endclass: consumer
```

Recommendation 4-124 —*Consumer transactors should use the* `vmm_channel::activate()`, `vmm_channel::start()`, `vmm_channel::complete()` *and* `vmm_channel::remove()` *methods to indicate the progress of the transaction execution.*

Using the `vmm_data::STARTED` and `vmm_data::ENDED` notifications require that the upstream transactor maintain a reference to the transactor descriptor instance while it flows through the channel and is executed by the consumer transactor. The *active slot* interface lets a producer transactor query the execution progress of a transaction directly from the channel itself. Indicating the `vmm_data::STARTED` and `vmm_data::ENDED` notifications is also implicit when using the `vmm_channel::start()` and `vmm_channel::complete()` methods.

Example 4-68. Transaction Execution Using the Channel's Active Slot

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr;
         ...
         this.in_chan.activate(tr);
         this.in_chan.start();
         ...
         this.in_chan.complete();
         this.in_chan.remove();
      end
   endtask: main
```

```
   ...
endclass: consumer
```

Suggestion 4-125 —*Consumer transactors may add completion status information to the transaction descriptor.*

If the transaction descriptor has properties that can be used to specify completion status information, these properties may be modified by the consumer transactor to provide status information back to the producer transactor.

Example 4-69. Providing Status Information In a Transaction Descriptor

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         transaction tr
         ...
         this.in_chan.start(tr);
         ...
            tr.status = ...;
         ...
         tr.in_chan.complete();
         ...
      end
   endtask: main
endclass: consumer
```

Suggestion 4-126 —*Consumer transactors may attach completion status information to the `vmm_data::ENDED` notification.*

If the transaction descriptor does not have properties that can be used to specify completion status information, the consumer transactor can provide status information back to the producer transactor via the `vmm_data::ENDED` notification.

The additional status information is provided as a separate status descriptor, derived from `vmm_data`, and attached to the `vmm_data::ENDED` notification by the `vmm_channel::complete()` method.

Example 4-70. Returning Status Information Via the `ENDED` Notification

```
class transaction_resp extends vmm_data;
   ...
endclass: transaction_resp

class consumer extends vmm_xactor;
```

```
    ...
    virtual task main();
       ...
       forever begin
          transaction tr;
          ...
          this.in_chan.start(tr);
          ...
          begin: status
             transaction_resp tr_status = new(...);
             ...
             this.in_chan.complete(tr_status);
          end
          ...
       end
    endtask: main
endclass: consumer
```

Out-of-Order Atomic Execution Model

Transactors with an out-of-order atomic execution model execute individual transactions in a potentially different order than they were submitted. The order in which transactions are selected for execution is protocol-specific and outside the scope of this book. Such transactors use a *non-blocking* completion model. As illustrated in Figure 4-9, the execution thread from the producer transactor (depicted as a dotted line) is not blocked while the transaction descriptor flows through the channel and is executed by the consumer transactor. It is blocked only when the channel is full and unblocks as soon as it is empty, regardless of the completion of the transaction.

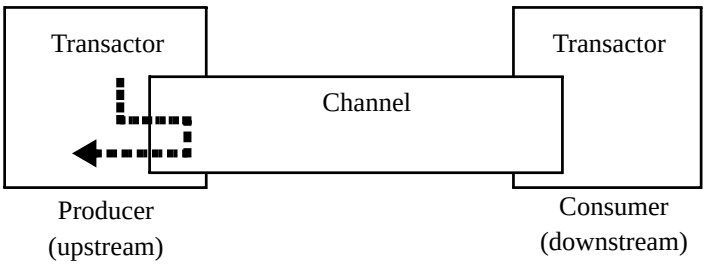


Figure 4-9. Non-Blocking Completion Model

The non-blocking completion model lets several transaction descriptors be submitted to the consumer transactor to be completed in the future. It is up to the producer

transactor to detect the completion of a transaction by waiting for the indication of the *vmm_data::ENDED* notification in the transaction descriptor or the *vmm_channel::ACT_COMPLETED* indication in the input channel, as illustrated in Example 4-71.

Example 4-71. Upstream of a Non-Blocking Completion Model

```
class producer extends vmm_xactor;
   ...
   virtual task main();
      ...
      ... begin
         transaction tr;
         ...
         out_chan.put(tr);
         fork
            begin
               automatic transaction w4tr = tr;
               w4tr.wait_for(vmm_data::ENDED);
               ...
            end
         join_none
         ...
      end
   endtask: main
   ...
endclass: producer
```

The suitability and proper implementation of this completion model requires that consumer transactors adhere to the guidelines presented next.

Recommendation 4-127 —*Input channel instances in consumer transactors should be reconfigured with a full level greater than one.*

The channel is responsible for eventually blocking the execution of the *vmm_channel::put()* method, not the downstream transactor. That blocking only happens if the channel is considered full. More than one transaction descriptor must be available in the channel to let out-of-order execution occur. If a full level of one is used, a blocking interface is created, and out-of-order execution is only possible if the downstream transactor implements additional transaction descriptor buffering internally.

Rule 4-128 — *A separate channel instance shall be used for each priority or class of service.*

Out-of-order transactors often execute transactions in a different sequence than submitted because they implement different priorities or class of services for different transactions. If a transactor offers more than one execution priority or class of service, it must use a different input channel for each. Using a single channel may block the execution of higher priority transactions because it is filled with low-priority transactions.

Rule 4-129 — *Consumer transactors shall use the* `vmm_channel::activate()`, `vmm_channel::start()`, `vmm_channel::complete()` *and* `vmm_channel::remove()` *methods to indicate the progress of the transaction execution.*

The `vmm_data::STARTED` and `vmm_data::ENDED` notifications require that the upstream transactor maintain a reference to the transaction descriptor while it flows through the channel and is executed by the downstream transactor. With an out-of-order execution model, it is a complex task to manage these references to all pending transaction descriptors and identify the next one that will be executed. The *active slot* interface lets an upstream transactor query the execution progress of a transaction directly from the channel itself.

Example 4-72. Out-of-Order Execution Using the Active Slot

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         ...
         this.in_chan.activate(tr, i);
         ...
         this.in_chan.start();
         ...
         this.in_chan.complete();
         this.in_chan.remove();
      end
   endtask: main
endclass: consumer
```

Suggestion 4-130 —*Consumer transactors may add completion status information to the transaction descriptor.*

If the transaction descriptor has properties that can be used to specify completion status information, these properties may be modified by the downstream transactor to provide status information back to the upstream transactor.

See Example 4-69 for an example.

Suggestion 4-131 —*Consumer transactors may provide completion status information via the* `vmm_channel::complete()` *method.*

If the transaction object does not have properties that can be used to specify completion status information, the consumer transactor can provide status information back to the producer transactor via the `vmm_data::ENDED` notification.

The additional status information is provided as a separate status descriptor, derived from `vmm_data`, and attached to the `vmm_data::ENDED` notification by the `vmm_channel::complete()` method.

See Example 4-70 for an example.

Non-Atomic Transaction Execution

Non-atomic transactors execute transactions in parallel, pipelined, through multiple attempts, or multiple partial sub-transactions or execute a transaction repeatedly at regular intervals. Such transactors use a *non-blocking* completion model. As illustrated in Figure 4-9, the execution thread from the upstream transactor (depicted as a dotted line) is not blocked while the transaction descriptor flows through the channel and is executed by the downstream transactor. It is blocked only when the channel is full and unblocks as soon as it is empty, regardless of the completion of the transaction.

The non-blocking completion model lets several transactions be submitted to the downstream transactor to be completed in the future. It is up to the upstream transactor to detect the completion of a transaction according to a mechanism defined by the downstream transactor.

The suitability and proper implementation of this completion model requires that the downstream transactor adheres to the guidelines presented next.

Recommendation 4-132 —*Input channel instances in the consumer transactors should be reconfigured with a full level greater than one.*

The channel instance is responsible for blocking the execution of the `vmm_channel::put()` method, not the downstream transactor. That blocking only happens if the channel is considered full. More than one transaction must be available in the channel to allow out-of-order execution to occur. If a full level of one is used, a blocking interface is created and non-atomic execution is only possible if the downstream transactor implements additional transaction descriptor buffering internally.

Rule 4-133 —*A separate channel instance shall be used for each priority or class of service.*

If a transactor offers more than one execution priority or class of service, it must use a different input channel for each. Using a single channel may block the execution of higher priority transactions because it is filled with low-priority transactions.

Rule 4-134 —*Consumer transactors shall use the `vmm_channel::get()` to immediately remove a transaction from the channel.*

Transactions in the channel are assumed to be available for execution. As soon as a transaction is selected for execution (either concurrently, partially or as the first instance of a recurrence), it must be immediately removed from the channel to prevent it from being selected again by another transaction execution thread.

Example 4-73. Removing a Transaction Descriptor from the Input Channel

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         ...
         this.in_chan.get(tr);
         tr.notify.indicate(vmm_data::STARTED);
         ...
      end
   endtask: main
endclass: consumer
```

Recommendation 4-135 —*Consumer transactors should indicate the* `vmm_data::STARTED` *and* `vmm_data::ENDED` *notifications.*

A producer transactor may track individual transactions by maintaining a reference to the transaction descriptors as they flow through the channel and are executed by the downstream transactor. Using the built-in indication of the execution of the transaction will eliminate the need for additional synchronization infrastructure in the upstream transactor.

Example 4-74. Indicating Transaction Execution Notifications

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         transaction tr;
         this.in_chan.get(tr, i);
         tr.notify.indicate(vmm_data::STARTED);
         ...
         tr.notify.indicate(vmm_data::ENDED);
      end
   endtask: main
   ...
endclass: consumer
```

The `vmm_channel::active()`, `vmm_channel::start()`, `vmm_channel::complete()` and `vmm_channel::remove()` methods cannot be used because they support an atomic—i.e., one at a time—execution model. When executing multiple transactions concurrently, these methods cannot be used.

Recommendation 4-136 —*An output "completion" channel should be used to send back (partially) completed transactions.*

A producer transactor may require information about the various intermediate completions of a transaction execution—each execution attempt, each sub-transaction and each occurrence of a recurring transaction. Since a transaction may have more than one completion indication, an output channel should be used to return completion information back to the producer transactor, as illustrated in Figure 4-10.

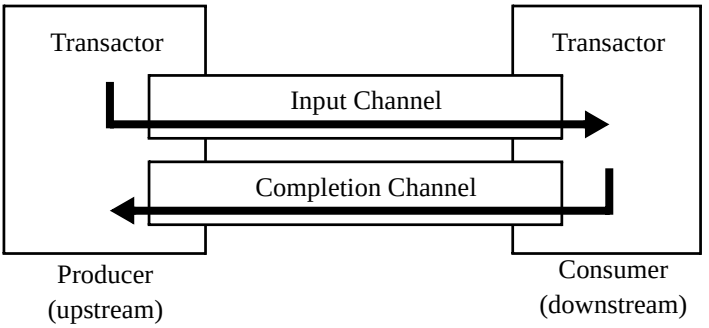


Figure 4-10. Completion Channel

Rule 4-137 — *Consumer transactors shall use the* `vmm_channel::sneak()` *method to add completed transaction descriptors to the completion channel.*

This usage will avoid the consumer transactor from stalling on a full completion channel, should the producer transactor fail to drain it. No data is lost even if the channel becomes full.

Example 4-75. Providing Completion Status Via Completion Channel

```
class consumer extends vmm_xactor;
   transaction_channel      in_chan;
   transaction_resp_channel compl_chan;

   virtual task main();
      ...
      forever begin
         ...
         this.in_chan.get(tr);
         tr.notify.indicate(vmm_data::STARTED);
         ...
         begin
            transaction_resp resp = new(...);
            tr.notify.indicate(vmm_data::ENDED, resp);
            this.compl_chan.sneak(resp);
         end
      end
   endtask: main
endclass: consumer
```

Suggestion 4-138 —*Consumer transactors may add completion status information to a copy of the transaction descriptor.*

If the transaction descriptor has properties that can be used to specify completion status information, these properties may be modified by the consumer transactor to provide status information back to the producer transactor.

A single transaction descriptor may result in multiple completion responses back through the completion channel. If the same instance is used, subsequent responses may modify the content of prior responses before the producer transactor has had time to process them. Using separate instances for each response will ensure that an accurate history of the transaction execution will be reported via the completion channel.

See Example 4-75 for an example.

Suggestion 4-139 —*Consumer transactors may use a different descriptor to return transaction completion information.*

If the transaction descriptor does not have properties that can be used to specify completion status information, the consumer transactor can provide status information back to the upstream transactor via a different status descriptor supplied through the completion channel.

The additional status information is provided as a separate descriptor, derived from `vmm_data`. A reference to the original transaction should be provided in the status descriptor. It is not necessary to overload all of the virtual methods in the status information class. See Example 4-75 for an illustration.

Passive Response

Passive transactors monitor transactions executed on a lower-level interface and report to the higher layers descriptions of the observed transactions. A passive transactor should report any protocol-level errors it detects, but the higher level transactors will be responsible for checking the correctness of the data carried by the protocol. As illustrated in Figure 4-11, passive transactors use an output channel to report transactions. Each observed transaction is reported using a new instance of the transaction descriptor.

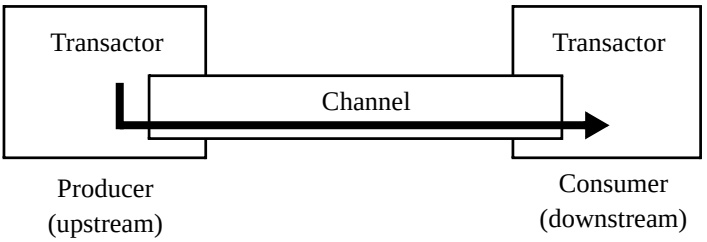


Figure 4-11. Passive Response Model

Note that the passive response model is not limited to passive transactors. It can be used to report on observed transactions in any other transactors. A reactive transactor may use the passive response model to report on the observed transactions that were actively replied to. A proactive transactor may use a passive response model to report on the received transactions as observed on a half-duplex interface.

The suitability and proper implementation of this response model requires that passive transactors adhere to the guidelines presented next.

Rule 4-140 — *Producer transactors shall put transaction descriptor instances in the output channel using the* `vmm_channel::sneak()` *method.*

The output channel will block the execution thread of the passive transactor if it ever becomes full. This blocking may break its implementation or cause data to be lost. The `vmm_channel::sneak()` method ignores the channel's full level and never blocks the execution thread of the upstream transactor. Because the passive monitor is observing the proper execution of a protocol, its execution should be regulated by the time required to execute a complete transaction.

Recommendation 4-141 — *Transactors should put an incomplete transaction descriptor instance in the output channel as soon as the start of a transaction has been identified.*

A consumer transactor may need to know when a transaction has started execution on an interface. For example, a half-duplex higher-level transactor would need to know if the transport medium is busy before attempting to execute its own transaction.

Waiting until the end of the transaction to put it in the output channel may make the information available too late.

Example 4-76. Incomplete Transaction Descriptor in an Output Channel

```
class producer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         ...
         tr = new;
         ...
         tr.notify.indicate(vmm_data::STARTED);
         this.out_chan.sneak(tr);
         ...
         tr.notify.indicate(vmm_data::ENDED);
      end
   endtask: main
endclass: producer
```

Rule 4-142 — *Transactors shall indicate the* `vmm_data::STARTED` *and* `vmm_data::ENDED` *notifications.*

This rule is a requirement of the previous guideline. If an incomplete transaction descriptor instance is put into the output channel, the higher-level transactor on the other side of the channel will need to know when the transaction has been completed. Using the built-in transaction completion notification event eliminates the need for additional synchronization infrastructure or mechanisms.

The timestamps associated with these notifications can also be used by the consumer transactors for identifying time-related information about the transaction, such as its total execution time.

Example 4-77. Monitoring Transactions from a Passive Transactor

```
class consumer extends vmm_xactor;
   ...
   virtual task main();
      ...
      while (1) begin
         ...
         this.in_chan.peek(tr);
         tr.notify.wait_for(vmm_data::ENDED);
         this.in_chan.get(tr);
         ...
      end
```

```
    endtask: main
endclass: consumer
```

Reactive Response

Reactive transactors monitor the transactions executed on a lower-level interface and may have to request additional data or information from higher-layer transactors to complete the transaction. Reactive transactors should report any protocol-level errors detected and locally generate protocol-level answers. But higher-level transactors are responsible for providing correct data content to be carried by the protocol.

As illustrated in Figure 4-12, reactive transactors use an output channel to request a transaction response. A second input channel is used to receive the transaction response to be applied to the lower-level interface. Each transaction response request is reported using a new instance of a transaction response descriptor object.

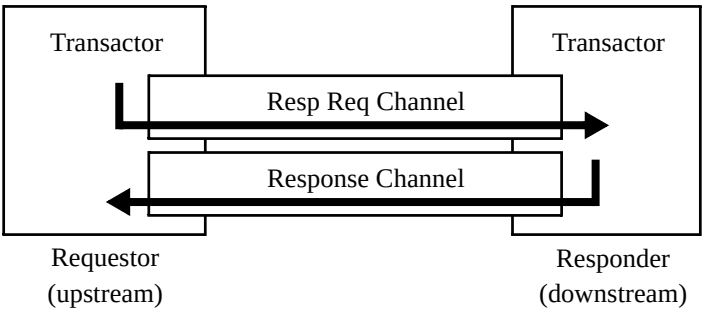


Figure 4-12. Reactive Response Model

Note that the reactive response model is only used to obtain higher-level data carried by the protocol. Where the entire set of possible responses are fully defined by the protocol, the response is internally generated by the reactive transactor. For example, deciding to reply to a USB transaction with an ACK, NACK or a STALL packet (or not replying at all) can be entirely decided internally. However, the content and length of a *DATA* packet in reply to an *IN* transaction should be provided by a reactive response model. Note that the response must be provided within sufficient time to avoid breaking the protocol.

The suitability and proper implementation of this response model requires that reactive transactors adhere to the guidelines presented next.

Rule 4-143 — *Requestor transactors shall use the* `vmm_channel::sneak()` *method to post a response request into the response request channel.*

The implementation of the protocol may require that the requestor transactor perform additional operations while the response is being "composed." The `vmm_channel::sneak()` method will ensure that the requestor transactor execution is never blocked, if only to immediately wait for a response via the response channel.

Example 4-78. Requesting a Response

```
class requestor extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         ...
         resp = new;
         ...
         this.req_chan.sneak(resp);
         ...
         this.resp_chan.get(resp);
         ...
      end
   endtask: main
endclass: requestor
```

Rule 4-144 — *Requestor transactors shall check that a response is provided within the required time interval.*

The time required to respond to a transaction is usually limited by the lower-level protocol specification. However, the time required to "compose" the response is controlled by the responder transactor. Thus, the requestor transactor can check only that the response comes back when required.

Example 4-79. Checking Response Request Fulfillment Delay

```
class requestor extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         ...
         resp = new;
         ...
         this.req_chan.sneak(resp);
         resp = null;
         fork
```

```
                this.resp_chan.get(resp);
                #(...);
             join_any
             disable fork;
             if (resp == null) ...
             ...
          end
       endtask: main
    endclass: responder
```

Recommendation 4-145 —*Requestor transactors should continue with a default response if no response is received after the maximum allowable time interval.*

To simplify the usage model of a reactive transactor, a default response may be used if a higher-level transactor failed to provide an explicit transaction response in time.

Recommendation 4-146 —*Requestor transactors should issue a warning message if no response is received after the maximum allowable time interval.*

The higher-level transactor may have wished to continue with the default response. Nonetheless, a message should be issued to inform the unwary user of a potential problem with the verification environment.

Recommendation 4-147 —*Transaction response request descriptors should solve to a valid random response when randomized.*

The content of a transaction response descriptor must be filled in by the responding reactive monitor. By default, a random—but valid—response should be provided. Therefore, the transaction response descriptor should be designed to provide a valid response when the *`randomize()`* method is used. The transaction response request descriptor could be user-extended to provide as more constrained response or procedurally filled in to provide a directed response.

Example 4-80. Providing a Random Response

```
class responder extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         this.req_chan.get(tr);
         ...
         tr.stream_id = this.stream_id;
         tr.data_id   = response_id++;
```

```
            if (!tr.randomize()) ...
            ...
            this.resp_chan.sneak(tr);
        end
    endtask: main
endclass: responder
```

Rule 4-148 — *Protocol-level response shall be randomly generated using an embedded generator.*

Protocol-level responses are fully defined by the protocol and can be selected by the reactive transactor without any input required from higher-level transactors. The response should be generated using an embedded factory-pattern generator. By default, the generator is constrained to produce the best possible response, but it can be unconstrained or modified to respond differently or inject errors.

Suggestion 4-149 — *Reactive transactors may randomly generate a default response using an embedded generator.*

To ease the creation of verification environments, a reactive transactor may be configurable to generate the complete protocol response internally instead of deferring the higher-level data to higher-level reactive transactors. A transactor that detects that a response was not provided within acceptable time and determines that the response request is still in the request channel, could assume that there are no higher-level transactors and choose to compose a default response on its own.

The response should be generated using an embedded generator. By default, the generator is constrained to produce adequate responses, but it can be unconstrained or modified to respond differently or to inject errors.

TIMING INTERFACE

Despite the rich set of completion models offered by the channel, it can only provide transaction information after-the-fact. A channel will transfer a data or transaction descriptor only once it has been completely received by a monitor. In some protocols or circumstances, higher-layer transactors require timing-related information as soon as that information is available, asynchronously from any transaction completion it may be associated with. For example, a MAC layer Ethernet transactor needs to know when the medium is busy so it can defer the transmission of any frame it may have. That information would be stale were it delayed until the frame occupying the medium had been completely received.

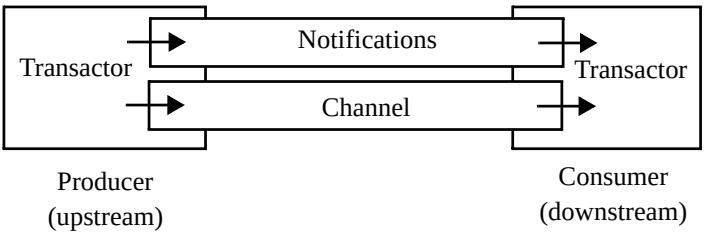


Figure 4-13. Notification Interface

Transaction-asynchronous timing information can be exchanged between two transactors via an instance of a notification service interface. Like a channel instance, a single instance is shared across two transactors. One transactor produces notification indications while the other waits for the relevant indications. A channel is used in parallel to transfer any transaction information once it is complete.

Rule 4-150 —*A `vmm_notify` extension shall be used to exchange notifications between two transactors.*

The extension of the `vmm_notify` class defines all notifications that can be exchanged between the two transactors.

Example 4-81. Notification Service Class

```
class eth_pls_indications extends vmm_notify;
   typedef enum {CARRIER, COLLISION} indications_e;

   function new(vmm_log log);
      super.new(log);
      super.configure(CARRIER, ON_OFF);
      super.configure(COLLISION, ON_OFF);
   endfunction: new
endclass: eth_pls_indications
```

Rule 4-151 — *References to notification service instances shall be stored in public class properties.*

This structure lets the connection between two transactors be made in arbitrary order. The first one creates the notification service instance, then the second one uses the reference to the instance in the first one.

Example 4-82. Notification Service Class Property

```
class eth_mac extends vmm_xactor;
   ...
   eth_pls_indications indications;
     ...
endclass: eth_mac
```

Recommendation 4-152 — *Notification service instances should be specified as optional constructor arguments.*

Like for channels, connecting two transactors requires that they share a reference to the same notification service instance. It should be possible to specify notification service instances to connect to as optional constructor arguments. If none are specified, new instances are internally allocated.

This connection requires that notification service instances be allocated if none are specified via the constructor argument list.

Example 4-83. Optional Notification Service Instances in Constructor

```
class eth_mac extends vmm_xactor;
   eth_pls_indications indications;
   ...
   function new(...
                eth_pls_indications indications = null);
      ...
      if (indications == null) indications = new(...);
      this.indications = indications;
      ...
   endfunction: new
   ...
endclass: eth_mac
```

Rule 4-153 — *A transactor shall not hold an internal reference to a notification service instance while it is stopped or reset.*

If a transactor holds a copy of the reference to a notification service instance in an internal variable, the notification service instance cannot be substituted with another one to modify the output or input of a transactor and dynamically reconfigure the

structure of a verification environment. While unavoidable during normal operations, a reset or stopped transactor should “release” all such internal references to let the notification service instance be replaced.

CALLBACK METHODS

The behavior of a transactor has to be controllable as required by the verification environment and individual testcases without requiring modifications of the transactor itself. These requirements are often unpredictable when the transactor is first written. By allowing the execution of arbitrary user-defined code in callback methods, transactors can be adapted to the needs of an environment or a testcase. For example, callback methods can be used to monitor the data flowing through a transactor to check for correctness, inject errors or collect functional coverage metrics.

Rule 4-154 — *Transactors shall have a rich set of callback methods.*

The actual set of callback methods that must be provided by a transactor is protocol-dependent. Subsequent guidelines will help design a suitable set in most cases. Additional callback methods should be provided as required by the protocol or the transactor implementation.

Recommendation 4-155 — *Transactors should call a callback method after receiving data, letting the user record, modify or drop the data.*

Whether it is a transaction descriptor or sampling a byte on a physical interface, the new input data should be reported to the user, via a post-reception callback method, to be recorded in or checked against a scoreboard, modified to inject an error or collect functional coverage metrics.

Recommendation 4-156 — *Transactors should call a callback method before transmitting data, letting the user record, modify or drop the data.*

Whether it is a transaction descriptor or driving a byte on a physical interface, the new output data should be reported to the user, via a pre-transmission callback method, to be recorded in or checked against a scoreboard, modified to inject an error or collect functional coverage metrics.

Recommendation 4-157 —*Transactors should call a callback method after generating any new information, letting the user record or modify the new information.*

Whenever a transaction requires locally generated additional information, the additional information should be reported to the user, via a post-generation callback method, to be recorded in or checked against a scoreboard, modified to inject an error or collect functional coverage. A reference to the original transaction should be provided to convey context information.

For example, a transactor prepending a packet with a preamble should call a callback method with the generated preamble data before starting the transmission process.

Recommendation 4-158 —*Transactors should call a callback method after making a significant decision but before acting on it, letting the user modify the default decision.*

Whenever a transactor makes a choice among several alternatives, the choice and available alternatives should be reported to the user, via a post-decision callback method, to be recorded in or checked against a scoreboard, modified to select another alternative or collect functional coverage. All information relevant to the context of the decision—candidates, rules and alternatives—should be provided to the user, along with the default decision, via the callback method.

For example, a transactor selecting traffic from different priority queues should call a callback method after selecting a queue based on the current priority selection algorithm but before pulling the next item from the selected queue. The user can then modify the selection.

Rule 4-159 —*All callback methods for a transactor shall be declared as virtual methods in a single class derived from* `vmm_xactor_callbacks`.

This declaration creates a façade for all available callback methods for a particular transactor. The common base class is required to be able to register the callback extension instances using the predefined methods and properties in the `vmm_xactor` class.

Example 4-84. Declaring a Façade of Callback Methods

```
virtual class mii_mac_layer_callbacks
         extends vmm_xactor_callbacks;
   virtual task pre_frame_tx(...);
   endtask
   ...
```

```
    virtual function void post_frame_rx(...);
    endfunction
    ...
endclass: mii_mac_layer_callbacks
```

Rule 4-160 — *Callbacks shall be declared as* `tasks` *or* `void functions`*.*

If the transactor implementation or protocol can support delays in the execution of a callback, it should be declared as a task. Callbacks that must be non-blocking must be declared as a function.

Restricting callback functions to `void` functions avoids difficulties with handling a return value from a function when multiple callback extensions are registered and cascaded in a transactor. Any status information returned from a callback method (such as a flag to indicate whether to drop the transaction) should be returned by modifying an instance referred to by an argument or a scalar argument passed by reference.

Rule 4-161 — *Arguments that must not be modified shall have the* `const` *attribute.*

Not all callback arguments can be modified. Some must not be modified because doing so would break the implementation of the transactor. Others should not be modified to avoid creating inconsistencies within the transaction being executed or observed.

Arguments without the `const` attribute can be modified by the user to inject errors.

Rule 4-162 — *A reference to the calling transactor shall be included in the callback arguments.*

This inclusion will let one extension of the callback methods be registered with more than one transactor instance and identify which transactor has invoked the callback method.

Rule 4-163 — *Transactors shall use the* `‘vmm_callback()` *macro to invoke the registered callbacks.*

Callback registration is implemented in the `vmm_xactor` base class. But calling the registered callback extensions is the responsibility of the transactor, extended from the base class. To remove the transactor implementation from the details of callback

registrations, and to ensure that they are called in the proper registration sequence, this macro must be used to invoke the callbacks.

Example 4-85. Invoking Registered Callback Extensions

```
virtual class mii_mac_layer_callbacks
         extends vmm_xactor_callbacks;
   virtual task pre_frame_tx(mii_mac_layer   xactor,
                             const eth_frame fr,
                             ...);
   endtask
   ...
endclass: mii_mac_layer_callbacks

class mii_mac_layer extends vmm_xactor;
   ...
   virtual task main();
      ...
      forever begin
         ...
         this.tx_channel.activate(fr);
         ...
         `vmm_callback(mii_mac_layer_callbacks,
                       pre_frame_tx(this, fr, ...));
         ...
      end
   endtask: main
endclass: mii_mac_layer
```

AD-HOC TESTBENCHES

The testbench infrastructure presented so far is quite flexible, scalable and powerful. It makes maximum use of the object-oriented features of SystemVerilog to leverage predefined functionality and create reusable verification components. The simulation steps in dynamic verification environments are well-defined.

However, not all testbenches need to leverage all of that flexibility, scalability and reusability. It must be possible to quickly construct ad-hoc testbenches that will be thrown away once they have helped accomplish their immediate purpose. Ad-hoc testbenches are quickly put together, typically by design engineers, to create some simple stimulus to verify the basic operation of a design unit. The response is usually checked visually.

In such simple testbenches, there is no need for complete management of the simulation steps, for dynamically reconfigurable transactors, for multiple sub-layers

of functional transactors, for integrating a self-checking structure, for sampling functional coverage nor for randomly generating high volume of data. Using VMM-compliant transactors to create these simple testbenches may be perceived as overkill and will require the acquisition of additional knowledge that is not typically required for RTL design.

Recommendation 4-164 —*Command-layer transactors should have an alternative encapsulation in a* `module`.

According to Rule 4-91, transactors are implemented in `classes`. Connecting a class to a module implementing a DUT requires the instantiation of an interface, the mapping of the wires in that interface to the wires in the DUT interface, the instantiation of the transactor configuration descriptor and finally the construction of the transactor. While they should remain implemented as classes, having the transactor class pre-instantiated in a module will simplify the instantiation process.

Example 4-86. Module Encapsulation for Transactor Class

```
module mii_mac_bfm(...);
...
parameter stream_id = -1;
program xactor;
   mii_cfg cfg;
   mii_mac_layer xact;

initial
begin
   string instance;
   $sformat(instance, "%m");
   cfg = new;
   xact = new(instance, stream_id,
              cfg, sigs.mac_layer);
   xact.start_xactor();
   ...
end
endprogram: xactor
endmodule: mii_mac_bfm
```

Rule 4-165 —*The encapsulation module shall be named the same as the encapsulated transactor class with a "_bfm" suffix.*

Class names and module names may share the same `$root` name space. To avoid collision, they must be named differently. Using the same name for both, with the addition of a suffix, clearly identifies the relationship between the class and the module.

Rule 4-166 — *The encapsulated transactor shall be started.*

The transactor should execute transactions without any further interventions. It must thus be started by calling its `vmm_xactor::start_xactor()` method in an `initial` block in the `program` block that instantiates it. Transactions will not actually execute until submitted via the input channel or the procedural interface.

Recommendation 4-167 — *The encapsulation module should have a parameter for the stream identifier and all properties in the transactor configuration descriptor.*

Modules are configured via parameters, not configuration descriptors.

Rule 4-168 — *The instance name of the transactor class instance shall be specified as the module instance name.*

The module instance name can be obtained using the `%m` format specifier in the `$sformat` task.

Recommendation 4-169 — *The module should use individual signals as its ports.*

This style may increase the number of port connections required when instantiating the module, but may ease the mapping of the signals to other signals. Furthermore, the module ports can enforce or document signal directions. The module port signals must be mapped to the interface instance inside the module.

Example 4-87. Individual Signals as Module Port

```
module mii_mac_bfm(output [3:0] txd,
                   ...
                   input        tx_clk,
                   ...);
mii_if sigs();
assign txd         = sigs.txd;
...
assign sigs.tx_clk = tx_clk;
...
endmodule: mii_mac_bfm
```

Alternative 4-170 — *The module may use the `interface` as its unique port.*

This style may minimize the number of port connections required when instantiating the module, but may require the mapping of the interface signals to other signals should the DUT or top-level module not use a compatible `interface`.

Example 4-88. Interface as Module Port

```
module mii_mac_bfm(mii_if sigs);
...
endmodule: mii_mac_bfm
```

Recommendation 4-171 —*An instance of the relevant atomic generator should be co-encapsulated.*

By including an atomic generator connected to the transactor in the encapsulation, it will be possible to easily create random stimulus by simply starting the generator and transactor. Because the transactor is always started by default, starting and stopping the generator will be sufficient to control the flow of random transactions.

Example 4-89. Co-Encapsulation of a Generator

```
module mii_mac_bfm(...);
parameter stream_id = -1;
...
program xactor;
   mii_cfg cfg;
   mii_mac_layer xact;
   eth_frame_atomic_gen src;

initial begin
   string instance;
   $sformat(instance, "%m");
   cfg = new;
   xact = new(instance, stream_id,
              cfg, sigs.mac_layer);
   xact.start_xactor();
   src = new(instance, stream_id, xact.tx_chan);
   ...
end
endprogram
endmodule: mii_mac_bfm
```

Rule 4-172 —*An instance of the relevant protocol checker shall be co-encapsulated.*

The protocol checker can be connected to the module ports or interface and verify the correctness of the protocol. This encapsulation will eliminate having to instantiate the checker separately from the transactor.

Recommendation 4-173 —*A procedural transaction-level interface should be provided.*

To further reduce the dependency on the object-oriented approach used by the data and transaction descriptor approach, a procedural interface should be provided. Designers should already be familiar with this kind of transaction-level interface as it has been used with Verilog for many years.

The tasks implementing the procedural interface simply create the corresponding transaction descriptor, randomizing any unspecified properties, then inject the transaction descriptor using the *`inject()`* method of the co-encapsulated generator.

It is important that the procedural interface executes in the reactive region to avoid race conditions.

Example 4-90. Transaction-Level Procedural Interface

```
module mii_mac_bfm(...);
...
parameter stream_id = -1;
...
program xactor;
   integer tx_count = 0;
   ...
   task tx_frame(input bit [47:0] da;
                 input bit [47:0] sa;
                 input bit [15:0] len);
      eth_frame fr = new;
      bit       ok;

      fr.stream_id = stream_id;
      fr.data_id   = tx_count++;
      ok = fr.randomize() with {
             dst == da;
             src == sa;
             len_typ == len;
             format == UNTAGGED;
          };
      if (!ok) begin
         `vmm_error(...);
         return;
      end
      src.inject(fr);
   endtask: tx_frame
   ...
endprogram: xactor
endmodule: mii_mac_bfm
```

Rule 4-174 — *Module-encapsulated transactors shall be instantiated in the top-level module.*

Modules cannot be instantiated in a class. Therefore, they cannot be used in an environment built from the *vmm_env* base class. They must be instantiated in the top module that also instantiates the DUT. This structure creates a verification environment identical to those that were created using Verilog or VHDL. The module-encapsulated transactor instances can be access via absolute hierarchical references.

Example 4-91. Instantiating a Module-Encapsulated Transactor

```
module top;
...
mii_mac_bfm mac(...);
eth_mac  dut(...);
...
endmodule: top
```

Example 4-92. Accessing an Instance of a Module-Encapsulated Transactor

```
program test;
initial begin
   ...
   top.mac.xactor.tx_frame(...);
   ...
end
endprogram
```

LEGACY BUS-FUNCTIONAL MODELS

Dealing with legacy bus-functional models is the opposite problem of packaging transactors for ad-hoc testbenches. Bus-functional models exist as *modules* with procedural transaction-level interfaces.

VMM-Compliance Upgrade

The ideal situation would be to modify the bus-functional model to make it VMM-compliant. This compliance requires potentially significant changes to the original source code. Assuming the bus-functional model is relatively well structured, the following list describes the steps that must be taken to make it VMM-compliant:

1. Encapsulate the ports of the *module* into an *interface*. Define a *modport* to match the ports on the original module. Rule 4-4 to Rule 4-9 are applicable. Do not use *clocking* blocks for synchronous signals.

2. Encapsulate all configuration parameters in a transactor configuration descriptor. Rule 4-104 to Recommendation 4-107 are applicable.
3. Transform module-level declarations into class properties and module tasks and functions into methods.
4. Encapsulate the body of *initial* blocks in tasks. Call these tasks in the constructor and in the extension of the *vmm_xactor::reset_xactor()* task. Rule 4-96 and Rule 4-97 are applicable.
5. Encapsulate the body of *always* blocks in a *forever* loop in tasks. Fork calls to these tasks in the extension of *vmm_xactor::main()*. Rule 4-93 to Rule 4-95 are applicable.
6. Create a transaction descriptor that can describe calls to and returned information from each transaction interface procedure. Rule 4-54 to Recommendation 4-86 are applicable. Define a channel of that class and instantiate in the transactor as input and/or output channels. Rule 4-111 to Recommendation 4-113 are applicable.
7. For driver operations, fork a thread in the extension of *vmm_xactor::main()* to pull transaction descriptors out of the input channel, interpret them, and call the corresponding transaction interface procedure. Rule 4-118 to Suggestion 4-139 are applicable.
8. For monitor operations, fork a thread in the extension of *vmm_xactor::main()* to call the corresponding transaction interface procedures, create the corresponding transaction descriptor, then sneak it into the output channel. Rule 4-140 to Recommendation 4-146 are applicable.
9. Create a callback façade class and add appropriate callback methods. Recommendation 4-155 to Rule 4-163 are applicable.

VMM-Compliant Interface

The transformation may be made more complex if the structure of the original code is poor. Investing in a complete rewrite may be wise in the long term. If rewriting the bus-functional model is not an option and the *module* cannot be turned into a *class*, it will be necessary to create a companion *class* that will interface directly with an instance of the *module*. Of course, because of the static nature of *module* instances, the companion *class* will effectively have a static instance as well.

The following list describes the steps required to create a VMM-compliant interface to a legacy bus-functional model implemented in a *module*:

1. Create a suitable transaction and transactor configuration descriptors. Rule 4-54 to Recommendation 4-86 and Rule 4-104 to Recommendation 4-107 are applicable.

2. Do not create an interface for the physical signals. Those are handled by the legacy *module*.
3. Create a transactor class, derived from *vmm_xactor*. The constructor should have arguments for all input and output channels and configuration descriptor but not for a virtual interface. Do not include an argument for the instance name either. Rule 4-89 to Recommendation 4-113 are applicable.
4. In the constructor, use the hierarchical name of the module instance as the transactor instance name. Set the values of the appropriate class properties in the configuration descriptor using the corresponding parameter values in the module instance. The parameters will have to be referenced using an absolute hierarchical name.

Example 4-93. Interfacing to a Single Instance of a Lefacy BFM

```
class utopia_mgmt extends vmm_xactor;
   utopia_mgmt_cfg cfg;
   ...
   function new(...);
      super.new("Utopia Mgmt", "top.mgmt", ...);
      cfg.is_intel = !top.mgmt.BusMode;
      ...
   endfunction: new
   ...
endclass: utopia_mgmt
```

5. Implement the required functionality to translate to/from transaction descriptors from/to the legacy bus-functional model in one or more independent threads forked off the extension of the *vmm_xactor::main()* task. Interface to the module instance using absolute hierarchical names to the necessary variables, tasks or functions.

It is necessary to use absolute hierarchical references into the *module* instance to interface the dynamic companion *class* instance to the static *module* instance. The steps described above use absolute hierarchical names hard-coded in the companion *class* implementation. This convention will work if a verification environment contains a single instance of the *module* at the specified hierarchical location. If multiple module instances are required or if the hierarchical location of the instance is not known a priori, this approach will not work.

SystemVerilog does not support dynamically computed hierarchical reference names. Any hierarchical reference must be static and known at compile time. Thus, a *string* cannot be used to simply store the path to the module instance. The solution is to easily replicate a *class* with different hard-coded hierarchical references using

a macro. Each macro expansion will yield a different *class*, designed to interface with a specific instance of its companion *module*.

Example 4-94. Companion Class Macro

```
‘define utopia_mgmt(path) \
class \path.utopia_mgmt extends utopia_mgmt; \
   ... \
   function new(...); \
      super.new("path", ...); \
      super.cfg.is_intel = !path.BusMode; \
      ... \
   endfunction: new \
   ... \
endclass
```

It is a good idea to separate instance-generic functionality from instance-specific functionality in a base class. *virtual* methods in that base class should be provided for each procedural interface available in the legacy *module*. The base class may also contain generic functionality, such as the channel-based transaction interface.

Example 4-95. Mapping Virtual Tasks to Instance-Specific Module Tasks

```
‘define utopia_mgmt(path) \
class \path.utopia_mgmt extends utopia_mgmt; \
   ... \
   virtual task read(input  [11:0] radd, \
                     output [ 7:0] rdat); \
      \path.read(radd, rdat); \
   endtask: read \
   ... \
endclass
```

Note how, in Example 4-94, an extended identifier is used to create a unique *class* name based on the hierarchical reference it is associated with. It is possible to interface to multiple instances of the *module* by creating a different class for each instance then instantiating them.

Example 4-96. Instantiating Instance-Specific VMM-Compliant Interfaces

```
module tb_top;
...
utopia_mgmt_bfm host0(...);
utopia_mgmt_bfm host1(...);
...
endmodule

program test;
   ‘utopia_mgmt(tb_top.host0);
```

```
    `utopia_mgmt(tb_top.host1);

    class tb_env extends vmm_env;
       utopia_host host[2];
       ...
       virtual function void build();
          \tb_top.host0.utopia_mgmt host0 = new(...);
          \tb_top.host1.utopia_mgmt host1 = new(...);
          host[0] = host0;
          host[1] = host;
          ...
       endfunction
       ...
    endclass: tb_env
endprogram
```

SUMMARY

This chapter has focused on the building blocks necessary to create verification environments. The layered architecture creates opportunities for higher levels of abstraction. It also encourages the decoupling of independent protocol functionality to promote reuse between block-level and system-level environments and between transaction-level and implementation-level verification. A message service and stimulation step management classes were described to standardize the look and feel of message and simulation controls.

A data-flow-based architecture for transactors, using a well-defined transaction interface object, has been defined. This architecture allows the creation of truly layered environments. It also allows the creation of higher-level verification components than the traditional bus-functional models available today. By breaking the close ties to a physical interface, transactors can now be written to operate—and be reused—at logical protocol layers.

This chapter concluded by presenting techniques and guidelines for dealing with legacy issues. The first legacy aspect that was considered is the use and integration of VMM-compliant transactors in more traditional testbench structures and verification expertise. This chapter also considered how to migrate or interface existing bus-functional models to make them usable in a VMM-compliant verification environment.

CHAPTER 5 STIMULUS AND RESPONSE

The verification planning process, described in Chapter 2, produced three distinct sets of requirements: functional coverage, stimulus generation and response checking. This chapter focuses on the last two sets of requirements. Functional coverage will be addressed in the following chapter.

The first part of this chapter will be of interest to those responsible for creating testcases. Testcases are created through directed or random stimulus. Directed stimulus can be considered as a subset of random stimulus and, with a properly designed random generator, is simple to create. Random generators must be designed to exercise the DUT according to the requirements outlined in the verification planning process (Chapter 2). Random generators must also be controllable to cover the entire spectrum of randomness between pure random and directed stimulus.

The second part of this chapter will be of interest to the verification engineers responsible for defining the verification environment and how the correctness of the design will be ascertained. The self-checking mechanism of a verification environment is highly DUT-specific. It is based on the response validation requirements defined during the verification planning process (Chapter 2).

GENERATING STIMULUS

The generation of data (packets, frames, instructions) or transaction descriptors is modeled separately from the data models themselves because of the different dynamics of their respective lifetimes. In a typical simulation, there will be thousands of data items or transaction descriptors created, flowing through transactors, recorded

and compared in the self-checking structure. On the other hand, there will be only a handful of data and transaction sources that need to exist at the beginning of the simulation and remain in existence until the end.

Generation can be a manual—or directed—process, where transaction descriptors and data items are individually created and submitted to the appropriate transactor. Generation can also be automated with the use of independent random generators. Using *randomness* is an approximation of automation: left to run for long enough, a random source will eventually generate, on its own, the stimulus necessary to exercise a large portion of the functionality to be verified. Unlike the typewriting monkeys of Shakespeare fame, random generators succeed in their task within a reasonable amount of time. They are not asked to replicate the exact directed stimulus an engineer would write to exercise a specific feature. Rather, random generators are expected to hit any one of a large number of features through non-optimal random stimulus sequences.

However, pure random stimulus—even constrained to be valid—is rarely useful. Shakespearian monkeys are unlikely to succeed in their task within a reasonable time because they produce random letter sequences. If they were restricted to producing letter sequences that create random English words within grammatically correct sentences, their chance of success would be dramatically improved. Similarly, generating Ethernet frames with random destination addresses is unlikely to verify the functionality of a device that has a specific address—except for the address matching functionality. It is also similarly unlikely that a random instruction generator will generate a well-formed loop structure that will also trigger a PC-match debug function in the processor. The degrees of freedom in random stimulus must be defined up front to create a mix of random but interesting scenarios.

Although the majority of verification engineers are more familiar with directed stimulus than random stimulus, random stimulus will be presented first. It is difficult to evolve from a directed stimulus process to an automated, random stimulus one. However, directed stimulus can be considered a subset of—or a highly constrained—random stimulus. The use of directed stimulus on an environment or in a process that is designed for random stimulus is therefore a natural evolution.

Rule 5-1 — *Verification environments shall be designed with random stimulus.*

A random-based verification environment can be constrained or overridden to produce directed stimulus. Accomplishing the opposite is much more difficult. If the directed stimulus only concerns a subset of the input paths to the DUT, the random stimulus on the other input paths can be used to provide background noise.

Random Stimulus

Random stimulus is traditionally used to generate background noise. It can also be used to generate the main test stimulus. This is often implemented as a “random” testcase, separate from the directed testcases, when time allows. Random stimulus can be used in lieu of directed stimulus to implement the bulk of the testbenches. Coupled with functional coverage to identify if the random stimulus has exercised the required functionality, constraints are used to direct the generation process in appropriate corner cases.

This section specifies guidelines on how to write autonomous generators that will create a stream of random data or transaction descriptors. Generators should be designed to be easily externally constrained, without requiring modifications of their source code. Constrained-random tests are then written, not by writing a completely new or slightly modified generator, but by adding constraints and scenario definitions to the reusable generators that already exist.

Predefined atomic and scenario generators are provided in the VMM Standard Library. As per Recommendation 5-23 and Recommendation 5-30, the *`vmm_atomic_gen()`* (see page 415) and *`vmm_scenario_gen()`* (see page 418) macros can be used to automatically create generators that follow all guidelines outlined in this section for any user-defined type derived from the *`vmm_data`* class.

Rule 5-2 — *A generator shall be modeled as a transactor.*

As such, all guidelines applicable to transactors are applicable to generators, unless explicitly superseded in this section.

Example 5-1. Generator are Transactors

```
class eth_frame_gen extends vmm_xactor;
   ...
endclass: eth_frame_gen
```

Rule 5-3 — *A generator shall have an output channel for each output stream.*

A generator is a transactor with no inputs. It produces streams of data or transaction descriptors that will need to be executed by transactors. To be able to connect the output of a generator to the input of a transactor, they must use the same transaction interface mechanism.

If a generator produces concurrent stimulus for multiple streams, it must have an output channel for each of the output streams. This channel will let each stream be connected to their respective execution transactors.

Rule 5-4 — *The reference to the generator output channels shall be in public class properties.*

This structure will allow several important operations that are required to implement testcases or build verification environments to be available, such as:

1. The channel can be queried, controlled or reconfigured.
2. The channel can be referenced as the input channel for a downstream transactor.
3. The channel can be replaced if dynamic environment reconfiguration is required.

Example 5-2. Generator Output Channel class Property

```
class eth_frame_gen extends vmm_xactor;
   ...
   eth_frame_channel out_chan;
   ...
endclass: eth_frame_gen
```

Rule 5-5 — *A reference to pre-existing output channel instances shall be optionally specifiable to the generator constructor.*

If no channel instance is specified, then the output channel is instantiated in the constructor. If a channel is specified, then its reference is stored in the appropriate public class property.

Example 5-3. Connecting a Generator to a Specified Channel Instance

```
class eth_frame_gen extends vmm_xactor;
   eth_frame_channel out_chan;
   ...
   function new(...,
                eth_frame_channel out_chan = null);
      ...
      if (out_chan == null) out_chan = new(...);
      this.out_chan = out_chan;
      ...
   endfunction: new
   ...
endclass: eth_frame_gen
```

Connecting a generator to a transactor requires that the output channel of the generator be the input channel of the downstream transactor. This connection can be accomplished only if they share references to a single channel instance.

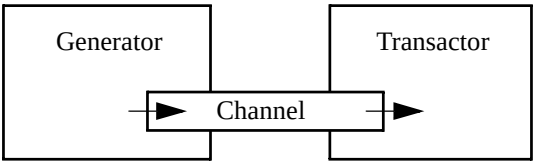


Figure 5-1. Connecting a Generator to a Transactor

The steps to connect a generator to a transactor are first, connect one of them internally to instantiate its channel, then second, pass a reference to that channel to the constructor of the other one.

Example 5-4. Instantiating the Generator First

```
class tb_env extends vmm_env;
   ...
   eth_frame_gen gen;
   eth_mac       mac;
   ...
   function void dut_env::build();
      this.gen = new(...);
      this.mac = new(..., this.gen.out_chan);
   endfunction: build
endclass: tb_env
```

Example 5-5. Instantiating the Transactor First

```
class tb_env extends vmm_env;
   ...
   eth_frame_gen gen;
   eth_mac       mac;
   ...
   function void dut_env::build();
      this.mac = new(...);
      this.gen = new(..., this.mac.tx_chan);
   endfunction: build
endclass: tb_env
```

Alternatively, a stand-alone channel can be instantiated then passed to the constructor of the generator and the transactor.

Example 5-6. Instantiating the Channel First

```
class tb_env extends vmm_env;
   ...
   eth_frame_channel gen_to_mac;
   eth_frame_gen     gen;
   eth_mac           mac;
   ...
   function void dut_env::build();
      eth_frame_channel gen_to_mac = new(...);
      eth_frame_gen     gen = new(..., this.gen_to_mac);
      eth_mac           mac = new(..., this.gen_to_mac);
   endfunction: build
endclass: tb_env
```

Rule 5-6 — *A generator shall randomize a single instance located in a public class property, then copy the final value to a new instance.*

This process is called a *factory pattern* and yields the most controllable generator. See section titled "Controlling Random Generation" on page 227 for the various constraint control mechanisms that can be used to control this generator pattern.

Example 5-7. Factory Pattern

```
class eth_frame_gen extends vmm_xactor;
   ...
   eth_frame randomized_fr;

   function new(...)
      ...
      this.randomized_fr = new;
   endfunction: new

   ...
      while (...) begin
         eth_frame fr;
         if (!this.randomized_fr.randomize()) begin
            `vmm_error(...);
             continue;
         end
         $cast(fr, this.randomized_fr.copy());
         ...
         out_chan.put(fr);
      end
   ...
endclass: eth_frame_gen
```

OOP Primer: Factory Pattern

A factory creates objects. That can be accomplished using *new*, but how can different objects (e.g., subjected to different constraints) be created without having to modify the original source code?

```
while (...) begin
   eth_frame fr = new;
   fr.display();
end
```

The factory pattern defers the object creation to a factory instance by using a *virtual* method (see “OOP Primer: Virtual Methods” on page 126). Instead of using *new*, virtual methods *allocate()* or *copy()* are used.

```
class eth_frame;
  ...
  virtual function
      eth_frame allocate();
    eth_frame fr = new;
    allocate = fr;
  endfunction
endclass
```

```
class generator;
  eth_frame factory;
  ...
    while (...) begin
      eth_frame fr;
      fr = factory.allocate();
      fr.display();
    end
  ...
endclass
```

The factory instance is publicly available to be replaced according to the needs of the application:

```
class my_eth_frame extends eth_frame;
   virtual function eth_frame allocate();
      my_eth_frame fr = new;
      allocate = fr;
   endfunction
endclass

program test1;
   generator gen = new;
initial begin
   my_eth_frame my_fr = new;
   gen.factory = my_fr;
   ...
end
endprogram
```

Recommendation 5-7 — *The name of the class property containing the randomized instance should have the prefix "`randomized_`".*

This naming convention will make it easier to identify the location, name and type of all randomized instances in a verification environment. It also clearly identifies the purpose of the class property.

Rule 5-8 — *The return value of the `randomize()` method shall be checked and an error be reported if it is false.*

If a contradiction in a set of constraints makes it impossible for the solver to find a solution, the *`randomize()`* method returns non-zero. It is important that an error be reported to indicate the problem with the constraints in the status of the simulation and to prevent a partial solution from being used.

Example 5-8. Checking the Success of Randomization Process

```
if (!this.randomized_fr.randomize()) begin
   `vmm_error(this.log, "Unable to find a solution");
   continue;
end
```

Rule 5-9 — *The value of the `stream_id` class property of the generator shall be assigned to the `stream_id` class property in the randomized instance before each randomization.*

The values should be set before every randomization attempt to ensure that the user does not accidentally modify the stream identifier in the randomized instance. It also ensures that the stream identifier will be set consistently should the randomized instance be substituted with another one to modify the constraints.

Example 5-9. Setting the *`stream_id`* Class Property

```
while (...) begin
   ...
   this.randomized_fr.stream_id = this.stream_id;
   ...
   if (!this.randomized_fr.randomize()) ...
   ...
end
```

The stream identifier class property is defined in the *`vmm_data`* base class and is inherited by all data and transaction descriptor classes. It is used to specify stream-specific constraints when constraints are added using a mechanism that is global to all instances.

Example 5-10. Specifying Constraints on a Subset of Streams

```
constraint eth_frame::tc1 {
   ...
   if (stream_id == 2) {
      ...
   }
}
```

Directed Stimulus

Directed stimulus is manually crafted to verify a specific feature of the design or to hit a specific functional coverage point. Not all of the stimulus needs to be directed. Random values can be used to fill portions of the stimulus that are not directly relevant to the feature being exercised. For example, the content of a packet payload is irrelevant to the correctness of the packet routing—only that it be transferred unmodified. Similarly, the content and identity of the general purpose registers used in an *ADD* instruction is not relevant, as long as the destination register eventually contains the accurate sum of the values contained in the two source registers.

Random stimulus may also be used as background noise on the interfaces not directly related to the feature being verified. The directed stimulus is focused on the interfaces directly implicated in the verification of the targeted functionality. Similarly, directed stimulus may be injected in the middle of random stimulus. This sequence may help identify problems that may not be apparent should the directed stimulus always be applied from the reset state.

Rule 5-10 — *Generators shall be stopped while directed stimulus is being injected.*

Directed stimulus is meant to replace random stimulus, not intermix with it. If the random generator is still running while directed stimulus is injected into its output stream, the resulting stimulus sequence will be unpredictable.

Generators may be stopped for the duration of the simulation while others, providing background noise, may be running as usual. Generators may be stopped at some points during the simulation, then restarted after the directed stimulus has been injected.

Example 5-11. Stopping a Generator at the Beginning of a Simulation

```
program test_directed;
...
initial begin
   ...
   env.start();
```

```
   env.host_src.stop_xactor();
   env.phy_src.stop_xactor();
   fork
      directed_stimulus;
   join_none
   env.run();
end

task directed_stimulus;
   ...
endtask: directed_stimulus
endprogram: test
```

Rule 5-11 — *Generators shall provide a procedural interface to inject data or transaction descriptors.*

Directed stimulus can be specified by manually instantiating data and transaction descriptors, then setting their properties appropriately. When injected in the output stream, the data or transaction descriptor is first subjected to the callback methods before being added to the generator output channel. The procedure returns when the directed data has been consumed by the output channel.

Example 5-12. Directed Transaction Interface

```
class eth_frame_gen extends vmm_xactor;
   eth_frame_channel out_channel;
   ...
   task inject(eth_frame fr,
               ref bit dropped);
      dropped = 0;
      `vmm_callback(eth_frame_gen_callbacks,
                    post_inst_gen(this, fr, dropped));
      if (!dropped) this.out_chan.put(fr);
   endtask: inject
endclass: eth_frame_gen
```

With this injection method available, directed stimulus can be easily created and injected into the output stream of the generator.

Example 5-13. Injecting a Directed Sequence

```
task directed_stimulus;
   eth_frame to_phy, to_mac;
   ...
   to_phy = new;
   to_phy.randomize();
   ...
   fork
```

```
            env.host_src.inject(to_phy, dropped);
            begin
               // Force the earliest possible collision
               @ (posedge tb_top.mii.tx_en);
               env.phy_src.inject(to_mac, dropped);
            end
         join
         ...
         -> env.end_test;
      endtask: directed_stimulus
```

Recommendation 5-12 —*Directed stimulus should not be directly added to the public output channel.*

Directed stimulus could be easily introduced in the output stream of the generator by directly putting instances of transaction descriptors in the output channel. This stimulus introduction would be accomplished by calling the *`vmm_channel::put()`* method directly. It requires that the directed stimulus be familiar with the transactor completion model to identify when the transaction execution has been completed. Furthermore, such stimulus would not be subjected to the callbacks methods of the generator and may not be recorded by the scoreboard or the functional coverage model.

This mechanism should only be used if it is necessary to create an out-of-order or partial-execution directed stimulus and should be avoided as much as possible. The reference to the output channel of a generator is public to allow for dynamic reconfiguration of an environment and to connect it to a downstream transactor. Its primary purpose is not to allow direct injection of directed stimulus.

Generating Exceptions

By default, transactors will execute transactions without errors, as fast as possible. However, the verification of a design requires that the limits of a protocol be stretched—and sometimes broken. A verification environment and the transactors that compose it must provide a mechanism for injecting exceptions in the execution of a transaction.

Recommendation 5-13 —*Exceptions should be described separately from transactions.*

Whether directly or randomly injected, exceptions should not be determined when a transaction descriptor is generated. Although quite feasible for transactions that are generated immediately upstream of a transactor—where there is a one-to-one correspondence between what is generated and what is executed—it becomes

impossible to control when the generator is removed from the transactor that must inject the exception through multiple transactor sub-layers or when the same transaction can be executed by different transactors, each with different possible exceptions.

For example, a TCP packet that is transmitted to a design via an MII interface must first be encapsulated into an IP packet, then segmented into one or more IP segments. Each segment is then encapsulated in a MAC frame. Each frame is them transmitted—and potentially retried—on the MII interface. How can an exception on nibble number 131 on the second attempt of the MAC frame carrying the last segment of the TCP packet be controlled? Rather, exceptions should be determined within the transactor responsible for injecting them.

The callback mechanism, as described in “Callback Methods” on page 198, can be used to cause a transactor to deviate from its default behavior. Within a callback, protocol exceptions—such as extra delays, negative replies or outright errors—can be injected without modifying the original transactor. Many exceptions can be defined and implemented in the callback methods themselves, such as inserting delays or corrupting the information in the transaction descriptor. Some exceptions must be implemented in the transactor itself, such as ignoring an entire transaction or prematurely terminating a transaction. In the latter case, callback methods will provide the necessary control mechanism to trigger them.

Directed exception injection is performed by extending the appropriate callback for the appropriate transactor within the testcase implementation. An instance of the callback extension is then prepended to the appropriate transactor callback registry. As shown in Example 5-14, a directed testcase uses the callback mechanism to force a collision on all input ports of an Ethernet device by aligning the transmission of the next frame in all MII transactors.

Example 5-14. Aligning the Transmissions in all MII Transactors

```
class align_tx extends mii_mac_layer_callbacks;
   local int waiting = 0;
   local int until_n = 1;
   local event go;
   ...
   virtual task pre_frame_tx(...);
      waiting++;
      if (waiting >= until_n) ->go;
      else @(go);
      waiting--;
   endtask: pre_frame_tx
enclass: align_tx
...
```

```
program test;
initial begin
   dut_env env = new;
   align_tx cb = new(...);
   env.build();
   foreach (env.mii[i]) begin
      env.mii[i].prepend_callback(cb);
   end
   env.run();
end
endprogram
```

Random stimulus is proving to be a powerful mechanism to improve the productivity of functional verification. But stimulus means more than primary data and transactions. It also includes protocol exceptions. Instead of having to explicitly inject protocol exceptions using a directed approach, these exceptions can be included randomly.

Rule 5-14 — *A randomized exception descriptor shall be used to randomly inject exceptions.*

Random injection of a protocol exception is accomplished by randomly generating an *exception descriptor.* That exception descriptor is implemented and generated using the same technique as transaction descriptors. Example 5-15 shows an exception descriptor for an MII MAC-layer transactor that can be used to create collisions.

Example 5-15. Exception Descriptor for an MII Protocol

```
class mii_mac_collision;
   typedef enum {NONE, EARLY, LATE} kind_e;
   rand kind_e       kind;
   rand int unsigned on_symbol;
        int unsigned n_symbols;

   constraint early_collision {
      if (kind == EARLY) on_symbol < 112;
   }
   constraint late_collision {
      if (kind == LATE) {
         on_symbol >= 112;
         on_symbol < n_symbols;
      }
   }
   constraint no_collision {
      kind == NONE;
   }
endclass: mii_mac_collision
```

If more than one exception can be injected concurrently during the execution of the transaction, the exception descriptor should properly model this capability.

Rule 5-15 — *An exception descriptor shall have a reference to the transaction descriptor it will be applied to.*

This reference will allow the expression of constraints to correlate protocol exceptions with the transactions they are applied to.

Example 5-16. Exception Descriptor for an MII Protocol

```
class mii_mac_collision;
   ...
   eth_frame frame;
   ...
endclass: mii_mac_collisions
```

Rule 5-16 — *An exception descriptor shall have a constraint block to prevent the injection of exception by default.*

To prevent the injection of protocol exception, an exception descriptor must be able to describe a *no-exceptions* condition as shown in Example 5-15. A constraint block should ensure that, by default, no exceptions are injected. Most of the testcases are not interested in exceptions and thus will use the transactor as-is. For the few tests responsible for verifying the response of the design to protocol exception, they simply need to turn off the constraint block.

Example 5-17. Enabling the Injection of Protocol Exceptions

```
program test_collisions;
...
initial begin
   ...
   env.build();
   ...
   env.phy.randomized_col.
      no_collision.constraint_mode(0);
   ...
   env.run();
end
endprogram: test_collisions
```

Rule 5-17 — *The exception descriptor shall be randomized using a factory pattern.*

The random exception generation may be built in the transactor itself, as shown in Example 5-18. However, this usage requires that the author of the transactor plans for

every possible exception that could be injected. If the source code for the transactor is available, the kinds of exceptions that can be injected by the transactor can be evolved according to the needs of the projects, even a truly reusable transactor should never have to be modified. If the source code is not available, it may be difficult to introduce additional exceptions in the transactor without introducing disjoint control mechanisms.

Example 5-18. Exception Generation Built into a Transactor

```
class mii_phy_layer extends vmm_xactor;
   virtual mii_if.phy_layer sigs;
   ...
   mii_phy_collision randomized_col;

   function new;
      ...
      this.randomized_col = new;
   endfunction: new
   ...
   task tx_driver();
      ...
      if (!randomized_col.randomize()) ...
      ...
   endtask: tx_driver
endclass: mii_phy_layer
```

The random exception generation can also be built into a callback extension. This mechanism can be used to add exception injection capabilities into a transactor that does not already support them, or this mechanism can be used to supplement the exceptions already provided by the transactor. Example 5-19 shows how the exception generation is built into a callback extension. The factory pattern requires that the environment maintains its own reference to the callback extension instance to let testcases control the randomized descriptor.

Example 5-19. Exception Generation in a Callback Extension

```
class gen_rx_errs extends mii_phy_layer_callbacks;
   mii_rx_err randomized_rx_err;
   ...
   virtual task pre_frame_tx(...);
      ...
      if (!randomized_rx_err.randomize()) ...
   endtask: pre_frame_tx

   virtual task pre_symbol_tx(...);
      if (this.randomized_rx_err.on_symbol == nibble_no)
         err = 1'b1;
```

```
    endtask: pre_symbol_tx
endclass: gen_rx_errs
```

Embedded Stimulus

Stimulus is generally understood as being applied to the external inputs of the design under verification. However, limiting stimulus to external interfaces only may make it difficult for some testcases to be performed. If the verification environment does not have a sufficient degree of controllability over the design, much effort may be spent trying to create a specific stimulus sequence to an internal design structure because it is too far removed from the external interfaces. This problem is particularly evident in systems where internal buses or functional units may not be directly controllable from the outside.

Suggestion 5-18 — *Design components can be replaced by transactors.*

Transactors need not be limited to driving external interfaces. They can be used to replace an internal design unit and provide control over that unit's interfaces. The transaction-level interface of the embedded transactor remains externally accessible, making the replaced unit interfaces logically external. Monitors can be similarly used to replace slave devices, as described in section titled "Embedded Monitors" on page 32.

For example, an embedded ARM core could be replaced with an AMBA AHB Interface master transactor, as illustrated in Figure 5-2. The ARM Core design *`module`* is replaced with a different *`module`*—with the same name—that contains an instance of the verification *`interface`* properly mapped to the *`module`* ports, as shown in Example 5-20. The AMBA AHB Interface master transactor is then associated with that *`interface`* instance to drive the *`module`* ports, as shown in Example 5-21. Transactions would thus be generated not by executing instructions but by having the transactor execute transaction descriptors. Connectivity is preserved and verified because the transactor is inserted within the original design unit interface. The run-time of testcases is also improved because fewer lines of code are simulated, there is no need to fetch instructions and there is no object code being executed by the processor core.

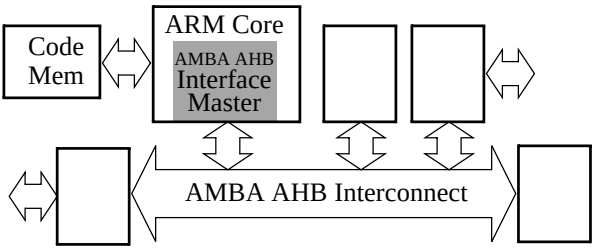


Figure 5-2. Replacing a Design Unit with a Transactor

Example 5-20. Replacement Module for Embedded Stimulus Generation

```
module arm_core(input  hclk,
                output mhbusreq,
                input  mhgrant, ...);

ahb_if sigs();
assign sigs.hclk    = hclk;
assign mhbusreq     = sigs.mhbusreq;
assign sigs.mhgrant = mhgrant;
...
endmodule
```

Example 5-21. Embedded Transactor

```
task dut_env::build();
   ahb_master core = new(...,
                         top.dut.core_i.sigs.master,
                         ...);
   ...
endtask
```

When substituting a design block for a transactor, it may be necessary to ensure that the generated stimulus is representative of system-level activity.

CONTROLLING RANDOM GENERATION

The objective of a random generator is to be able to create all of the necessary stimulus to fully verify a design. Some of that stimulus can be created without any constraints other than those required to ensure that valid stimulus is being created. Other stimulus will require additional or modified constraints to hit certain corner cases or inject errors.

The ability to create certain stimulus patterns is directly related to the ability to express the constraints that will cause the patterns to be generated. If it is not possible to express a constraint between two variables, it will not be possible to create a relationship between their respective values. Declarative constraints can be expressed only across properties (or sub-properties) of a class. Procedural constraints can be expressed across disjoint variables using the *std::randomize with* statement. Declarative constraints are preferable as they can be added to, modified or removed without modifying or duplicating the generation code.

Instead of coding a directed testcase to verify a particular function of the design, it may be simpler to modify the constraints on the generators to increase the likelihood that they will generate the required data streams on their own. Adding, or modifying constraints that already exist, can be simple if the guidelines outlined in this chapter have been followed.

Alternative 5-19 —*A testcase may turn the* *rand* *mode of properties ON or OFF.*

Because generators are always randomizing the same instance, it is possible to "remove" the *rand* mode on arbitrary properties which, for a particular test, must remain constant. The *rand* mode of some properties may have been turned off by default to prevent invalid data from being generated. Turning them back on and adding relevant constraints can be used to inject errors. This is a procedural constraint modification and can be executed at any time during the execution of a testcase.

Example 5-22. Controlling the *rand* Mode of a Class Property

```
this.host_src.randomized_obj.dst = this.cfg.mac.addr;
this.host_src.randomized_obj.dst.rand_mode(0);
this.host_src.randomized_obj.src = this.cfg.dut_addr;
this.host_src.randomized_obj.src.rand_mode(0);
```

Alternative 5-20 —*A testcase may turn constraint blocks ON or OFF.*

Because generators are always randomizing the same instance, it is possible to turn constraint blocks ON or OFF using the *constraint_mode()* method. This method is used to disable constraint blocks that may have been designed to prevent the injection of errors or to modify the distribution of the generated values to obtain a different distribution. This is a procedural constraint modification and can be executed at any time during the execution of a testcase.

Example 5-23. Controlling Constraint Blocks

```
program test_collisions;
...
initial begin
```

```
    ...
    env.build();
    ...
    env.phy.randomized_col.no_collision
      .constraint_mode(0);
    ...
    env.run();
end
endprogram
```

Alternative 5-21 —*Testcases may provide external* `constraint` *block definitions.*

If the definition of a randomized `class` contains `extern constraint` blocks, they can be defined for each testcase. This style requires the pre-existence of an undefined `extern constraint` block, as per Recommendation 4-86, and can only be used to add constraints. The new `constraint` block definition is added simply by including a source file that defines it. This change is a declarative constraint modification that applies to all instances of the `class`. They will be taken into consideration (unless the `constraint` block is turned OFF) whenever an instance of the `class` is randomized. The constraints apply for the entire duration of the testcase execution.

Example 5-24. Specifying External Constraints

```
program test;
...
constraint eth_frame::tc1 {
    data.size() == min_len;
}
initial begin
    ...
    env.run();
end
endprogram
```

Alternative 5-22 —*A testcase may replace randomized instances with instances of a derived class with additional constraints.*

It is not always possible to create the desired data stream simply by turning constraints on or off or by tweaking distribution weights. If the constraints or variable distribution weights did not exist prior, it will not be possible to create the necessary stimulus.

Because generators are always randomizing the same instance (“OOP Primer: Factory Pattern” on page 217), it is possible to replace the randomized instance with an instance of a derived class. And because the `randomize()` method is virtual, the additional or overridden constraint blocks in the derived class will be used. Unlike the

external constraint block implementation, this mechanism allows the addition of class properties and methods and further extension of virtual methods to facilitate the expression of the required constraints. It also allows the redefinition of existing constraint blocks and methods.

Although the class extension is declarative and global to a simulation, the substitution of the randomized instance with an instance of this new class is procedural. This constraint modification can be executed at any time during the execution of a testcase.

Example 5-25. Replacing a Factory Instance

```
program test_...;
...
class long_eth_frame extends eth_frame;
   constraint long_frames {
      data.size() == max_len;
   }
endclass: long_eth_frame
...
initial begin
   env.build();
   begin
      long_eth_frame fr = new;
      env.host_src.randomized_obj = fr;
   end
   ...
   top.env.run();
end
endprogram
```

Example 5-26. Constraining the Test Configuration

```
class duplex_test_cfg extends test_configuration;
   constraint test_Y {
      mode == DUPLEX;
   }
endclass

program test_Y;
initial begin
   duplex_test_cfg my_cfg = new;
   top.env.randomized_cfg = my_cfg;

   top.env.run();
end
endprogram
```

Atomic Generation

Atomic generation is the generation of individual data items or transaction descriptors. Each is generated independently of the items or descriptors that have been previously generated or that will be subsequently generated. Atomic generation is like using a random function that returns a complex data structure instead of a scalar value.

Example 5-27. Atomic Generator

```
class eth_frame_gen extends vmm_xactor;
   ...
   eth_frame randomized_fr;
   ...
   virtual protected task main();
      ...
      while (...) begin
         ...
         if (!this.randomized_fr.randomize()) ...
         ...
      end
      ...
   endtask: main
endclass: eth_frame_gen
```

Atomic generation is simple to describe and to use, as shown in Example 5-27. Its ease of use is the reason atomic generation is used to illustrate most of the generation and constraints examples in this book—and other literature. However, it is unlikely to create interesting stimulus sequences on its own, even with the addition of constraints.

For example, how could an atomic instruction generator be constrained to generate a well-formed loop structure? How about a nested loop structure? Generating interesting stimulus sequences requires the ability to constrain random stimulus within the context of the previous and subsequent items and descriptors.

Recommendation 5-23 —*The predefined atomic generator* `vmm_atomic_gen` *should be used.*

The predefined atomic generator created by the `vmm_atomic_gen` macro follows all relevant guidelines. With a few keystrokes, a powerful atomic generator can be created for any type derived from the `vmm_data` class. See “vmm_atomic_gen” on page 415 for more details.

Scenario Generation

A scenario generator generates sequences of data items or transaction descriptors. That is the purpose of the `randsequence` statement—also known as the *stream generator*. However, the `randsequence` statement is a procedural statement. Modifying the constraints or defining new scenarios in a scenario generator implemented using this statement requires that the generator itself be modified—potentially breaking or modifying the behavior of existing tests. A scenario generator that makes use of the object-oriented framework will allow such modifications to be made without modifying the generator.

A scenario is described—and randomized—using an array of data items or transaction descriptors. It is possible to constrain items and descriptors with respect to others before or after it in the sequence because the array is randomized all at once. This style lets the solver consider the past, current and future items and descriptors when solving a constraint set over a stimulus sequence.

Example 5-28. Scenario Generator

```
class eth_frame_sequence extends vmm_data;
   ...
   rand eth_frame items[];
   ...
enclass: eth_frame_sequence
...
class eth_frame_sequence_gen extends vmm_xactor;
   ...
   eth_frame_sequence randomized_sequence;
   ...
   virtual protected task main();
      ...
      while (...) begin
         if (!this.randomized_sequence.randomize()) ...
         ...
      end
   endtask: main
endclass: eth_frame_sequence_gen
```

The scenario generator shown in Example 5-28 is remarkably similar to the atomic generator shown in Example 5-27. In fact, they are identical. The only difference is in the type of the object being randomized. Whereas an atomic generator randomizes the data item or transaction descriptor directly, a scenario generator randomizes a scenario descriptor that contains the data items or transaction descriptors that form the scenario. All of the techniques illustrated with an atomic generator can be applied to a scenario generator because they are essentially the same thing. They only differ in the type of object they randomize.

Different scenarios are described as different variations of a scenario descriptor. Based on a (randomized) scenario identifier, each scenario is described as a series of constraints or procedural steps as shown in Example 5-30. The simplest scenario is one containing only one unconstrained item, effectively mimicking an atomic generator.

Example 5-29. Scenario Descriptor

```
class eth_frame_sequence;
   ...
   rand int unsigned sequence_kind;
   rand int unsigned length;

   rand eth_frame items[];
   ...
endclass: eth_frame_sequence
```

Example 5-30. User-Defined Scenarios

```
class my_scenarios extends eth_frame_sequence;
   ...
   constraint burst_of_short_frames {
      if (scenario_kind == SHORT_FRAMES) {
         length > 2;
         foreach (items[i]) {
            items[i].data.size == items[i].min_len;
         }
      }
   }
   ...
   virtual task apply(eth_frame_channel channel,
                      ref int           n_insts);
      if (scenario_kind == REPEAT_10) begin
         repeat (10) begin
            channel.put(items[0]);
         end
         n_insts = 100;
         return;
      end
      ...
      super.apply(ch, n_inst);
   endtask: apply
endclass: my_scenarios
```

The only problem is that the SystemVerilog standard does not define the semantics of randomizing the size of an array very precisely. In particular, what is the final content of a randomized array if the randomized size is greater than the original size of the array prior to randomization? For arrays of scalars, additional elements are added and are randomized. But what about arrays of *class* instances? Are new instances of the

class created then randomized? What if the array is of a virtual class that cannot be directly instantiated? What if the array should be filled with instances of derived classes?

The following guidelines will help implement a scenario generator that will be portable across different SystemVerilog simulators.

Recommendation 5-24 —*Scenario generators should be implemented using the* `vmm_scenario_gen` *macro.*

The `vmm_scenario_gen` macro is used to quickly implement scenario generators using a predefined set of base classes and templates. It ensures that they follow the guidelines outlined in this section and present a consistent user-extension interface.

Rule 5-25 — *Scenarios shall be described using extensions of a scenario descriptor base class.*

Additional scenarios are described as additional variants of the scenario descriptor. Different variants can be described in the same extensions, based on the value of the `scenario_id` attribute, or as concurrent extensions. The base class needs to provide scenario identification management and item allocation services.

Rule 5-26 — *A scenario shall be identified using an* `integer` *class property, initialized using a* `define_scenario()` *method.*

The `scenario_id` class property is an integer value identifying the specific scenario described by the descriptor. The actual value that corresponds to a particular scenario is not functionally relevant and assigned by the scenario descriptor base class in the `define_scenario()` method. This method ensures that scenario identifiers are unique, even if a new scenario is added to an intermediate extension of the class. It also manages the potential need for allocating the data or transaction descriptor instances in the scenario before randomization.

Example 5-31. Identifying a New Scenario

```
class my_scenarios extends eth_frame_sequence;
   int SHORT_FRAMES = define_sequence(...);
   ...
endclass: my_scenarios
```

Rule 5-27 — *A random scenario shall be specified using a constraint block with a conditional constraint based on the value of the* `scenario_id` *class property.*

Random scenarios are defined as constraints on the array of individual items that make up the scenario. To enable the co-existence of concurrent scenarios within the same descriptor as different values of the `scenario_id` class property, the constraints defining a particular scenario must be enclosed in a qualifying pre-condition. Using a separate constraint block for each random scenario will also allow a scenario to be redefined in a further extension of the scenario descriptor.

Example 5-32. Specifying a New Random Scenario

```
class my_scenarios extends eth_frame_sequence;
   ...
   constraint burst_of_short_frames {
      if (scenario_kind == SHORT_FRAMES) {
         ...
      }
   }
endclass: my_scenarios
```

Rule 5-28 — *Procedural scenarios shall be implemented as extensions of the scenario descriptor's* `apply()` *method.*

Some scenarios are better described using a procedural implementation—such as using a `repeat` loop to generate a long sequence of identical data or transaction descriptors or by using the `randsequence` statement. These procedural—and often directed—scenarios are then included in random tests as another available scenario. That way they will be surrounded by other random or procedural scenarios and may uncover an unexpected problem.

A procedural scenario may require runtime interaction with the environment to be fully specified. The data may be a priori generatable but may also depend on runtime feedback information from the environment. If visibility over the environment or design signals is required, the necessary signals can be accessed by using cross-module references.

Example 5-33. Defining a Procedural Scenario

```
class my_scenarios extends eth_frame_sequence;
   ...
   constraint back_off {
      if (sequence_kind == BACKOFF) length == 1;
   }
   ...
```

```
      virtual task apply(...);
         if (this.sequence_kind == REPEAT_10) begin
            repeat (10) begin
               channel.put(this.items[0]);
            end
            ...
            return;
         end
         if (this.sequence_kind == BACK_OFF) begin
            @ (posedge env.mii.crs);
         end
         super.apply(...);
      endtask: apply
   endclass: my_scenarios
```

Rule 5-29 — *Extensions of the scenario descriptor's `apply()` method shall not execute the default implementation if the scenario items are manually added to the output channel.*

The default implementation of the `apply()` method forwards the content of the generated scenario to the generator's output channel. If a procedural scenario leaves the scenario array non-empty and the default implementation is called, some items may be added twice to the output channel.

The default implementation of the method can be skipped by not calling `super.apply()` method in a scenario descriptor extension, as illustrated in Example 5-33, if the `REPEAT_10` scenario is generated.

Recommendation 5-30 —*The predefined scenario generator `vmm_scenario_gen` should be used.*

The predefined scenario generator created by the `vmm_scenario_gen` macro follows all relevant guidelines. With a few keystrokes, a powerful scenario generator can be created for any type derived from the `vmm_data` class. See "vmm_scenario_gen" on page 418 for more details.

Multi-Stream Generation

It may be necessary to coordinate the generation of scenarios on different stimulus streams to create interesting conditions in the design under test. Scenario generators generate stimulus of a single stream. It is not possible to express constraints across scenarios in different streams because they are generated in different generators and are not within the object being randomized by any given generator. The ability of

expressing constraints across streams requires that the data or descriptors for all of the streams be randomized as a single object.

Alternative 5-31 —*Multiple single-stream generators can be synchronized.*

This approach can be used if the multi-stream scenario simply requires the initial synchronization of normally-independent streams. It cannot be used if cross-correlation of the content of the individual streams or synchronization of individual transactions within the streams is required.

For example, to generate a collision on an Ethernet medium by forcing all devices to start transmitting at the same time can be accomplished by synchronizing all transmitting transactors. This can be implemented through callback extensions, as shown in Example 5-14.

Alternative 5-32 —*A single-stream scenario generator can generate a multi-stream scenario.*

A single-stream scenario generator only knows about its corresponding output channel. The individual transactions that make up the generated scenarios are applied to that channel. But that is only a *default* behavior. A scenario descriptor could refer to other output channels and its `apply()` method could forward the generated transactions to the various output channels to form a multi-stream scenario.

Example 5-34. Multi-Stream Scenario Descriptor

```
class my_scenarios extends eth_frame_sequence;
   ...
   virtual task apply(eth_frame_channel channel,
                      ref int           n_insts);
      if (this.sequence_kind == ...) begin
         wb_cycle cyc = new(...);
         ...
         channel.put(this.items[0]);
         channel.put(this.items[1]);
         ...
         env.host.exec_chan.put(cyc);
         ...
         fork
            channel.put(this.items[2]);
         join_none
         ...
         env.host.exec_chan.put(cyc);
         ...
         return; // As per Rule 5-29
      end
      super.apply(channel, n_insts);
```

```
      endtask: apply
   endclass: my_scenarios
```

Alternative 5-33 —*A multi-stream scenario generator can be used.*

A multi-stream scenario generator generates and concurrently applies a scenario for each one of its output streams. The individual output streams need not carry the same type of data or transaction descriptors. A multi-stream scenario is generated by randomizing a multi-stream scenario descriptor that constrains a scenario descriptor for each of the output streams, as shown in Example 5-35. A multi-stream scenario generator is similar to a single-stream scenario generator except that it has multiple output channels and, by default, applies each stream to their respective output channel concurrently.

Example 5-35. Multi-Stream Scenario Descriptor

```
class dut_ms_sequence;
   rand eth_frame_sequence to_phy;
   rand eth_frame_sequence to_mac;
   rand wb_cycle_sequence  to_host;
   ...
   virtual task apply(eth_frame_channel to_phy_chan,
                      eth_frame_channel to_mac_chan,
                      wb_cycle_channel  wb_chan);
      ...
   endtask
endclass: dut_ms_sequence
```

State-Dependent Generation

The traditional way to make the random generation process dependent on external state information is to embed knowledge of the external state data into the generator itself as procedural code. The external state is mirrored in the generator and is represented by a combination of variable values and current location of the various execution threads. Modifying the generation rules to reach new corner cases or inject errors required modification of the code itself.

The external controllability of the generators presented earlier was based on the declarative nature of constraints and methods involved in the randomization process. To maintain the same level of controllability in a state-dependent generator, it is necessary to make the relevant state data declaratively visible to the randomization process. They can then be externally modified to create new corner cases or inject errors without modifying the generator itself.

Randomization of a stimulus item—and the constraints considered while randomizing it—has a well-defined scope: the object instance where the *randomize()* method is called. To make the randomization process state-dependent, the state information must be brought into that scope. Unfortunately, the stimulus item is often not directly modifiable to make the necessary state information visible. It is typically a generic data model or transaction descriptor provided along with the transactors that should not be modified to include DUT-specific or test-specific concerns.

The alternative is to create a broader scope by composing the data model or transaction descriptor with all of the information required to express the constraints necessary to create corner cases or inject errors. This higher-level object is the one that is randomized instead of only the data or transaction. The randomization process now has the ability to include arbitrary state information and use that information declaratively to constrain the next stimulus item to be generated.

The additional state information can come from many different sources. Its nature is defined by the stimulus requirements of the DUT, to create valid stimulus according to higher protocol levels, reach corner cases and inject relevant error conditions. It may include—but is not limited to—DUT configuration information, interface signal values, internal state data, past stimulus items and protocol state descriptors.

The set of valid or interesting values may be limited by configurable parameters in the DUT. For example, a MAC interface will only accept frames that have a matching destination address. But the address of a MAC interface is programmable (otherwise, every Ethernet device that uses that MAC interface design would have the same address). When generating frames to be sent to a MAC interface, the destination address class property must be constrained to match (or not match to verify the proper rejection of frames addressed to another device).

This constraint can be accomplished by setting the destination address class property to the desired value and turning its `rand` attribute OFF, as shown in Example 5-36. Alternatively, an Ethernet frame extended with the MAC interface configuration can be randomized, as shown in Example 5-37.

Example 5-36. Configuration-Dependent Generation

```
this.host_src.randomized_obj.dst = this.cfg.mac.addr;
this.host_src.randomized_obj.dst.rand_mode(0);
this.host_src.randomized_obj.src = this.cfg.dut_addr;
this.host_src.randomized_obj.src.rand_mode(0);
```

Example 5-37. Configuration-Dependent Generation using Composition

```
class dut_eth_frame extends eth_frame;
   test_cfg  cfg;

   constraint match_dut_address {
      if (!cfg.mac.promiscuous) dst == cfg.dut_addr;
   }
   constraint valid_src_address {
      src == cfg.mac.addr;
   }
   ...
endclass: dut_eth_frame
```

This latter approach has the advantage that the address-matching constraint can be arbitrarily overloaded to implement different configuration-dependent generation rules, without modifying the generator itself. For example, a certain number of frames with mismatching addresses can be generated as shown in Example 5-38.

Example 5-38. Modifying Configuration-Dependent Generation

```
class my_dut_eth_frame extends dut_eth_frame;
   rand bit match;

   constraint match_dut_address {
      match dist {1:/9, 0:/1};
      if (match) dst == cfg.dut_addr;
      else       dst != cfg.dut_addr;
   }
   ...
endclass: my_dut_eth_frame
```

If the validity of a protocol requires a certain sequence in the values of data items transmitted back and forth between two peers, it is not possible to randomly generate individual data items without knowing where, in the protocol state sequence, the exchange is currently situated.

For example, the successful, error-free exchange of data between two peers on a TCP socket requires the following exchange of packets between the client and the server:

```
client   ---- SYN     --->   server
         <--- SYN+ACK ----
         ---- ACK     --->
         <--- ...
      (no SYN, no FIN, no RST)
               ...     --->
```

```
---- FIN     --->
<--- ACK     ----
<--- FIN     ----
---- ACK     --->
```

To complicate the problem, a TCP server must be able to maintain several concurrent connections with a certain number of clients, including multiple clients connected to the same server port and a port on a client connected to different ports on the server. The protocol state information must thus be tracked for several concurrent connections, each evolving independently. Random stimulus must be generated coherently on all concurrent connections, including creating new ones and closing existing ones.

Before designing and implementing a random generator, it helps to consider what type of stimulus it will have to generate to exercise the design under all interesting conditions and corner cases. If we only consider connection establishment, maintenance and closing on the TCP server, the generator must be able to create—on its own or with the help of constraints—the following interesting situations:

1. Maximum number of established sessions (one per client, with a single client)
2. Maximum number of ports open
3. Maximum number of established sessions on a single port (host and server)
4. Many very short sessions
5. Sessions to privileged and ordinary ports

The generator should be designed to be constrainable to reach these points if it does not on its own. The alternative would be to code a generator that can randomly select a specific point to reach, then generate the stimulus for it—but that is no different than writing directed testcases in a case statement with a random selector.

Rule 5-34 — *Actions relevant to advancing the state of the protocol shall be generated.*

Randomly generated stimulus must be of the proper level of abstraction to offer the necessary controls to achieve the desired outcome. By generating individual TCP packets, the level of abstraction is not high enough to reach the desired functional coverage goal—expressed in terms of connections, not packets. The stimulus generator should therefore generate connection-related actions, not individual packets. These actions will eventually map to individual packets, but the packet will be constrained by the desired action to meet the requirement of the higher level protocol.

The possible TCP connection actions that can be generated are: initiate a new connection, maintain an existing connection or terminate an established connection. The action must also be targeted to a new client, an existing client or an existing connection. Once the client, connection and actions have been generated, that information can be used to generate the next TCP packet to be transmitted to the host.

Alternative 5-35 —*Stimulus may be the result of several randomization steps.*

The stimulus generation process may be easier to describe and constrain if decomposed into separate steps. For TCP packets, it is easier when two steps are used: First, a (random) decision is made whether to create a new client or use an existing one. Next, an action and the corresponding packet are randomly generated.

The TCP packet generator first randomizes a client selection descriptor that includes the necessary test configuration parameters to limit the number of clients, as shown in Example 5-39. If an existing client with existing connections is picked, one of the sessions is randomly selected for potential action.

Example 5-39. Randomized TCP Client Selector

```
class tcp_client_selector;
   int max_n_clients;
   int n_clients;

   rand bit new_client;
   ...
   constraint boundaries {
      if (n_clients >= max_n_clients) new_client == 0;
      if (n_clients == 0)             new_client == 1;
   }
endclass: tcp_client_selector
```

The TCP action (and corresponding packet) are generated in a second step, where a TCP action descriptor, shown in Example 5-40, is randomized. The descriptor includes the relevant client and session configuration information to express constraints that will generate a valid action and packet.

Example 5-40. Randomized TCP Action Descriptor

```
class tcp_client_action;
   tcp_client_session client;

   typedef enum {OPEN, CHAT, CLOSE} action_e;
   rand action_e action;

   constraint open_iff_before_established {
      if (client.state < tcp_client_session::ESTAB)
```

```
            action == OPEN;
         else action != OPEN;
      }
      ...
   endclass: tcp_client_action
```

Rule 5-36 — *State transition rules shall be specified using constraints.*

The traditional mechanism for describing state-dependent relationships and sequences is to use a `case` statement, with the current state register as a selector. Each state-dependent decision is implemented in the code block corresponding to the required current state value. Random decisions may be included in the process. For example, if there are three possible outcomes for a given current state, the actual outcome may be decided randomly amongst the three candidates. The problem with this approach is that the decision process or the constraints in random decisions can only be modified (for example, to inject errors) by modifying the generator itself. Because a `case` statement is procedural, not declarative, it cannot be replaced or overloaded using object-oriented programming techniques.

The next-state decision can be done using constraints. Because constraints are declarative, they can be externally modified or added to using any of the techniques shown earlier. Example 5-41 shows how the content of the next TCP packet is defined using constraints based on the TCP action to be performed and the current state of the target connection.

Example 5-41. Subset of Next-Packet Constraints

```
class tcp_client_action
   tcp_client_session client;
   ...
   rand tcp_packet pkt;
   ...
   constraint syn_iff_open {
      if (action == OPEN) pkt.syn == 1;
      else                pkt.syn == 0;
   }
   ...
   constraint fin_iff_close {
      if (action == CLOSE) pkt.fin == 1;
      else                 pkt.fin == 0;
   }
   ...
endclass: tcp_client_action
```

Rule 5-37 — *Individual state transition rules shall be specified as separate constraint blocks.*

Note how each next-state transition for the next-packet generation in Example 5-41 is coded in a separate constraint block, one per protocol rule. Constraint blocks are the smallest declarative constructs that can be modified or turned off. Expressing constraints with this level of granularity provides the same granularity of control for testcases over the protocol implementation. For example, a testcase could randomly reset a connection halfway though the synchronization steps by overloading the relevant constraint block, such as illustrated in Example 5-42.

Example 5-42. Injecting Protocol Errors

```
class my_action extends tcp_client_action
   constraint never_rst {
      if (action == OPEN) rst == 0
   }
endclass: my_action
```

Rule 5-38 — *The generator shall update the protocol state based on the response received.*

The TCP packet generator must also update the state of connections based on packets received from the server in response to packets sent by the client. An independent process can monitor the received packets and update the appropriate session descriptors, as shown in Example 5-43.

Example 5-43. Updating Connection State Based on DUT Response

```
if (pkt.rst) session.state = tcp_client_session::CLOSED;
else begin
   case (client.state)
   tcp_client_session::SYN_SENT:
      if (pkt.syn && pkt.ack)
         client.state = tcp_client_session::ESTAB;
      else client.state = tcp_client_session::CLOSED;
   ...
   endcase
end
```

Which Type of Generator to Use?

This chapter has presented several styles of stimulus generators, each with their capability and limitations. The question that naturally arises is: Which type of generator should be used? The guidelines below should help answer this question.

Rule 5-39 — *Atomic generators shall be used for configuration generation.*

Configuration information is generated only once at the beginning of a simulation. Therefore, the data-sequencing capabilities of a scenario or state-dependent generators are not required.

Atomic generators may also be used for simple data interfaces where it will not be necessary to create specific sequences of input stimulus.

Rule 5-40 — *Scenario or state-dependent generators shall be used initially.*

Whether a simple scenario generator may be used or a state-dependent generator must be used depends on the nature of the stimulus required by the design. If the stimulus must follow certain protocol states, then a state-dependent generator must be used.

They should be able to provide a large portion of the stimulus required by the design. It may even be possible that they will create more complex scenarios, on their own, potentially eliminating the need for multi-stream generators.

Rule 5-41 — *Scenario generators shall be replaced with a single multi-stream generator in an extension of the verification environment.*

A multi-stream generator may be required to exercise the design under certain conditions. It should be introduced into the environment by replacing the scenario generators for the streams now generated by the multi-stream generator. This substitution should be performed in an extension of the verification environment to avoid breaking any of the tests written using independent scenario generators. The substitution is performed in the extension of the `vmm_env::build()` method, by setting the reference to the individual scenario generators to `null` so they will be reclaimed by the garbage collector and using their output channels.

Example 5-44. Replacing Scenarios Generators with a Multi-Stream Generator

```
class my_dut_env extends dut_env;
   ms_scenario_gen gen;
   ...
   virtual function void build();
      super.build();
      gen = new(..., {this.src[0].out_chan,
                      this.src[1].out_chan,
                      this.apb_gen.out_chan}, ...);
      this.src[0]  = null;
      this.src[1]  = null;
      this.apb_gen = null;
   endfunction: build
```

```
endclass: my_dut_env
```

Rule 5-42 — *References to replaced scenario generators shall be set to* `null`.

By setting the reference to the individual scenario generators to `null`, they will be reclaimed by the garbage collector. But more importantly, it will prevent them from being started in the extension of the `vmm_env::start()` method, thus competing for the output channel.

SELF-CHECKING STRUCTURES

Because each self-checking structure is unique to the design being verified, there is little functionality that can be reused across self-checking structures. Hence, this book does not describe a base or utility class to help implement them. Common functionality—and reuse—can be found across projects in the same application domain. Entire self-checking structures may even be reusable across generations of the same product. But this book is concerned with multiple application domains and no specific products; hence, unlike the other portions of the verification environment, the book cannot present some base-level functionality.

This section does present techniques for encapsulating and integrating a self-checking structure as well as techniques for transaction-accurate response checking. These techniques are generic and are applicable to a wide range of domains.

Rule 5-43 — *The self-checking structure shall be encapsulated in a* `class`.

The `class` will encapsulate the self-checking structure independently of the nature of its actual implementation. If the self-checking structure is implemented using SystemVerilog, the implementation would be inlined or instantiated as required in the `class`. If the self-checking structure is implemented in C or another co-simulated model or is performed offline, then the `class` will abstract the integration mechanics to present a pure SystemVerilog access interface to the rest of the verification environment.

Encapsulating the self-checking structure in a `class` also enables it to be multiply instantiated, should a system be composed of multiple instantiations of the corresponding block.

Example 5-45. Self-Checking Structure

```
class scoreboard;
   ...
endclass: scoreboard
```

Rule 5-44 — *The self-checking structure shall require a reference to the test and design configuration descriptor in its constructor.*

The response to be checked will depend on the configuration of the design. Since that configuration is only known at the start of the *vmm_env::build()* method, it must be passed to the self-checking structure.

The constructor can then use the value of the configuration descriptor to instantiate the appropriate data structures or open the required files.

Example 5-46. Self-Checking Structure Configuration

```
class scoreboard;
   local test_cfg cfg;
   ...
   function new(test_cfg cfg);
      this.cfg = cfg;
      ...
   endfunction: new
   ...
endclass: scoreboard
```

Rule 5-45 — *The self-checking structure shall be instantiated in a public class property of the verification environment.*

Various components of the verification environments will need to have access to the self-checking structure, either to register stimulus, or to check response or to update design state information. The instance of the self-checking structure will be globally visible and accessible whenever the environment is visible.

Example 5-47. Self-Checking Structure Instantiation

```
class tb_env extends vmm_env;
   test_cfg cfg;
   ...
   scoreboard sb;
   ...
   function void build();
      super.build();
      ...
      this.sb = new(this.cfg);
      ...
```

```
      endfunction: build
      ...
endclass: dut_env
```

Rule 5-46 — *The self-checking structure shall have no reference to the verification environment.*

The self-checking structure needs information from all components in the verification environment. Thus, it is tempting to have the self-checking structure access the various components in the environment to obtain that information. Because the environment is built using a structure of class instances, not module instances, cross-module references—which are resolved after elaboration—cannot be used. Hierarchical class references—which are resolved at compilation time—must be used. This activity creates compilation dependencies between the self-checking structure and the verification environment.

Furthermore, if a self-checking structure contains internal references to the unit-level verification environment, it will not be reusable in the system-level environment.

Rule 5-47 — *The self-checking structure shall have a procedural interface to report injected stimulus, exceptions and observed responses.*

The verification environment will access the tasks in the self-checking structure `class` to record stimulus and response data as required. This access ensures a strictly linear compilation dependency between the environment and the self-checking structure.

It will also enable a unit-level self-checking structure to be reused in a system-level environment, in a different verification environment. The response for different instances of a design block is verified by accessing the procedural interface in the appropriate instance of the self-checking structure.

Example 5-48. Self-Checking Structure Procedural Interface

```
class scoreboard;
   ...
   function void sent_from_mac_side(eth_frame fr);
      ...
   endfunction: sent_from_mac_side
   ...
   function void received_by_mac_side(eth_frame fr);
      ...
   endfunction: received_by_mac_side
   ...
endclass: scoreboard
```

Recommendation 5-48 —*`Channels` and `mailboxes` should not be used to interface to the self-checking structure.*

Channels and mailboxes are designed to implement communication media between concurrent processes. A self-checking structure should be implemented using zero-delay processing and should not require running as a separate process.

If the self-checking structure requires simulation time to verify and observe response or to record a reported stimulus, the blocking thread should be forked from the procedural interface to prevent the threads of the transactors interfacing with the self-checking structure from blocking.

Scoreboarding

This section presents guidelines to help you implement a self-checking structure implemented in SystemVerilog using scoreboarding techniques.

Rule 5-49 — *The data structure that holds the expected response shall be designed to minimize the look-up operation.*

Because it may be difficult to predict the exact order in which outputs will be observed, it is unlikely that a single variable or queue will be sufficient to implement the data structure. At any given time, there may be several possible valid responses from the design, depending on the implementation details and internal resources usage. Response may be concurrently observed on multiple interfaces, with irrelevant —and unpredictable—ordering among them.

The comparison function will have to identify, as efficiently as possible, if the most-recently observed response matches one of the possible valid responses. Data structure design is beyond the scope of this book—and has been well-covered in many other works—but some language constructs and techniques can be used.

Recommendation 5-50 —*A `queue` should be used to store in-order responses.*

If the expected response must come in a specific order, a `dynamic array` or a `queue` should be used to store the individual responses in the order in which they are expected.

Response is predicted in the same order in which it will be observed. This prediction requires that a new prediction be added at the end of the data structure and observed response be checked against—and removed from—the front of the data structure. A `queue` is better suited because it is designed to offer the same performance when

adding or removing an element, regardless of its position in it. A *dynamic array* would need to have its entire content shifted by one position every time a response is checked. This shifting operation would be more inefficient, especially if there is a large number of predicted responses in the data structure.

Recommendation 5-51 —*Multiple queues should be used to store independent in-order streams.*

The response on several concurrent output interfaces may be observed in any order. But the sequence of responses observed on a single interface is likely to be in-order. Using one *queue* per output interface allows the *queue* to store the predicted response of each interface in the expected order. They also let the observed response across the various interfaces be efficiently checked in any order: It is supposed to match the predicted response at the front of the *queue* corresponding to the output interface where it was observed.

Example 5-49. Checking In-Order Response for Multiple Output Ports

```
class port_sb;
   eth_frame frame_seq[$];
endclass: port_sb

class scoreboard;
   port_sb predicted[4];
   ...
   function void received_frame(eth_frame fr,
                                int unsigned port_no);
      eth_frame expect =
         predicted[port_no].frame_seq.pop_front();
      if (!fr.compare(expect)) ...
   endfunction: received_frame
   ...
endclass: scoreboard
```

Suggestion 5-52 — *The scoreboard may not need to predict the exact transactions that were lost.*

Dropping transactions or data items is not a functional fault in many applications. In some classes of designs, it may be extremely difficult to predict the exact transactions or data items that will be lost or dropped by the design. The decision to drop a transaction is often the result of the state of the design, resource utilization and some implementation-dependent timing or discard algorithm.

Rather than trying to duplicate the discard decision and circumstances in the scoreboard, it may be easier to simply recognize when transactions may have been dropped, and assume that they have been properly dropped. The number of assumed-

dropped transactions is then confirmed using drop-count statistics in the design or some other means of confirming the assumption about dropped transactions.

A transaction may be assumed dropped when it is present in the data structure ahead of the predicted response that corresponds to the currently observed response. Similarly, any predicted response left in the data structure at the end of the simulation is either assumed to have been dropped or still in-process inside the design.

Example 5-50. Assuming Predicted Responses are Dropped

```
function void received_frame(eth_frame fr,
                             int unsigned port_no);
   forever begin: look_for_match
      eth_frame expect;
      if (predicted[port_no].frame_seq.size() == 0) ...
      expect = predicted[port_no].frame_seq.pop_front();
      if (fr.compare(expect)) break;
      maybe_dropped.push_back(expect);
   end
endfunction: received_frame
```

Suggestion 5-53 — *The scoreboard may not need to predict the response with full accuracy.*

In some designs, it may not be possible to predict the timing or value of the response with complete accuracy. For example, the result of a fixed-point operation in a DSP design may not exactly match the predicted response calculated using floating-point operations. Or the exact timing of a response may be implementation-dependent as long as it falls within a certain window.

The observed response may be compared against the predicted response and be considered valid as long as it is within an acceptable margin of error or bit error rate.

Example 5-51. Verifying the Result of a Fixed-Point Operation

```
function bit rcompare(real actual,
                      real expect,
                      real err)
   real delta;

   delta = actual - expect;
   if (delta < 0.0) delta= -delta;
   rcompare = delta <= err;
endfunction
```

Recommendation 5-54 —*Serial numbers should not be embedded in the stimulus data.*

To facilitate the checking of the response ordering or locating the corresponding expected response in the data structure, a unique serial or sequence number is often added to the data in user-specified fields. This numbering will make it impossible to reuse the self-checking structure in a system-level environment, should the user-specified field have to be used to transport higher-level data.

Suggestion 5-55 —*An index table may be useful to locate any matching predicted response.*

Instead of using serial numbers embedded in the stimulus data, it may be equally efficient to use a *hashing function* to create a likely-unique key from individual observed response values. That key can be used to look up an *associative array* that would provide an index or pointer to the actual location of the expected data. This strategy can eliminate costly search procedures throughout the data storage structure.

Example 5-52. Using a Hashing Function to Locate Expected Response

```
function int hash(eth_frame fr)
   hash = fr.compute_crc();
endfunction

class sb_where_to_find_frame;
   int unsigned port_no;
   int unsigned queue_no;
endclass: sb_where_to_find_frame

sb_where_to_find_frame index_tbl[int];

function bit check(eth_frame actual)
   sb_where_to_find_frame where;
   eth_frame              q[$];
   eth_frame              expect;

   check = 0;
   if (!index_tbl[hash(actual)].exists()) return;
   where = index_tbl[hash(actual)];
   q = sb.port[where.port_no].queue[where.queue_no];
   expect = q.pop_front();
   if (actual.compare(expect)) check = 1;
endfunction: check
```

Integration with the Transactors

Transactors must be written independently of the verification environment for a particular design if they are to be reusable in a different environment or with different designs. The self-checking structure for the design is also created separately to be reusable in a system-level environment. How can two independent components of the verification environment—the self-checking structure and a transactor—be integrated to work together to verify a particular design?

Alternative 5-56 —*The procedural interface of the self-checking structure can be called from a callback extension.*

If a callback method has all of the required stimulus, response or state information, it can pass that information to the scoreboard by invoking the appropriate procedure. As per Rule 4-37 on page 131, these callback extensions must be registered first.

Example 5-53. Integrating a Scoreboard via Callback Methods

```
class sb_mac_cbs extends eth_mac_callbacks;
   scoreboard sb;

   function new(scoreboard sb);
      this.sb = sb;
   endfunction: new

   virtual task
      post_frame_tx(eth_mac             xactor,
                    eth_frame           frame,
                    eth_frame_tx_status status);
      if (status.successful && !this.sb.rx_err) begin
         this.sb.sent_from_mac_side(frame);
      end
      ...
   endtask: post_frame_tx
endclass: sb_mac_cbs
...
class tb_env extends vmm_env;
   virtual function void build();
      ...
      begin
         sb_mac_cbs cb = new;
         this.mac.append_callback(cb);
      end
      ...
   endfunction: build
   ...
endclass: tb_env
```

Alternative 5-57 —*The procedural interface of the self-checking structure can be called by a thread draining an output channel.*

Any output channel must be drained to prevent the accumulation of data and the eventual blocking of the producer thread. If the information in the transaction or data descriptors coming out of the channel is sufficient for the scoreboard, the thread used to drain the channel can be used to call the appropriate procedure in the scoreboard.

Note that this mechanism cannot be used if the flow of descriptors is required by an upstream higher-layer transactor.

Example 5-54. Scoreboard Integration by Draining an Output Channel

```
virtual function void build();
   ...
   fork
      forever begin
         eth_frame fr;
         this.mac.rx_chan.get(fr);
         this.sb.received_by_phy_side(fr);
       end
   join_none
   ...
endfunction: build
```

Alternative 5-58 —*The procedural interface of the self-checking structure can be called by a thread waiting for the status of an indicated* `vmm_notify` *notification.*

If a transactor provides all required information with the status of a notification, a thread can forward that information to the scoreboard by calling the appropriate procedure. Note that the status of an indicated notification is lost when the subsequent indication occurs. Thus, it is important that the scoreboard integration thread does not block before waiting for the next indication of the notification.

Example 5-55. Scoreboard Integration by Waiting on Notification

```
virtual function void build();
   ...
   fork
      forever begin
         eth_frame_tx_status fr;
         this.mac.notify.wait_for(this.mac.FRAME_SENT);
         $cast(status,
               this.mac.notify.status(
                  this.mac.FRAME_SENT));
          if (status.successful) begin
             this.sb.sent_from_phy_side(status.fr);
```

```
            end
          end
        join_none
    end
```

Dealing with Exceptions

Protocol exceptions should be injected by the transactor that generates the protocol in question. This approach avoids complicating the data generation process and makes it possible to inject errors in a lower-level protocol where there may not be a one-to-one mapping between the high-level data item and the low-level transactions transporting it—or low-level maintenance transactions that do not even involve high-level data.

To be able to correctly predict the response, the self-checking structure must be made aware of exceptions injected during the application of stimulus to the design. The exception may be benign—such as the insertion of wait states—or the exception may cause a higher-level transaction to be repeated—such as a collision on an Ethernet frame—or the exception may cause the entire high-level data transported by the low-level transaction to be dropped—such as unsuccessful USB transaction in a bulk transfer.

The task of predicting the consequence of such exceptions, even if incrementally injected by the various transactor layers, is not more difficult than if the exact same information had been entirely generated up front. The self-checking structure should maintain context information for each transaction. That information should be readily available via the context and implementation references in the data and transaction descriptors, as recommended in Recommendation 4-64 on page 146. The self-checking structure can thus determine the integrity of a high-level transaction carried by a low-level transaction that was subjected to a particular exception.

Rule 5-59 — *A description of the injected exception shall be reported to the self-checking structure.*

If exception injection is built into the transactor, the mechanism for reporting the exceptions depends on the notification mechanism provided by the transactor. If the exception injection is implemented as a user-extension of a callback method, it should be reported via the same callback extension that records stimulus with the self-checking structure.

Example 5-56. Reporting Exceptions to the Self-Checking Structure

```
class scoreboard;
   ...
   bit rx_err;
```

```
   ...
endclass: scoreboard
...
class gen_rx_errs extends mii_phy_layer_callbacks;
   ...
   virtual task pre_symbol_tx(...);
      if (...) begin
         err = 1;
         this.sb.rx_err = 1;
      end
   endtask: pre_symbol_tx
endclass: gen_rx_errs
```

Rule 5-60 — *Error injection callback registration shall be prepended to the self-checking integration callback registration.*

Exceptions can be injected through a user-defined extension of a callback method. For example, it could corrupt the CRC of a frame to be injected. For the self-checking structure to be aware of those exceptions, the execution of the exception injection callback must be performed before the self-checking integration callback.

To ensure the proper callback execution order, self-checking integration callback extensions must be registered first. Exception injection callback extensions must then be registered using the *`vmm_xactor::prepend_callback()`* method.

Example 5-57. Prepending Error Injection Callbacks

```
env.build();
begin
   gen_rx_errs cb = new;
   env.phy.prepend_callback(cb);
end
```

SUMMARY

This chapter has presented an architecture and associated coding guidelines for modeling reusable and controllable random generators. These generators can also be used as an insertion point for directed stimulus. If designed properly, generators can be controlled using a variety of ways: making random variables constant, turning off constraint blocks to relax constraints, adding external constrain block definitions to tighten constraints and using inheritance to modify constraints.

This chapter has also presented techniques for encapsulating and integrating self-checking structures. Techniques that can be used to implement a self-checking structure using scoreboarding have also been detailed.

CHAPTER 6 COVERAGE-DRIVEN VERIFICATION

Traditionally, coverage is used as a confidence-building metric. It is used as a safety net to ensure that the verification plan was as complete and that the design was verified as thoroughly as possible. Coverage measurements are done toward the end of the verification process, when the bulk of the testcases have been written. The coverage metrics are usually expected to be initially relatively high, demonstrating that the test suite, painfully developed over the previous months, is effective at verifying the complete functionality of the design. The remainder of the project is spent either tweaking the test suite to increase the coverage numbers and to justify why some of these numbers cannot—or need not—reach 100%.

But what if the coverage metrics are, unexpectedly, initially low? This means that a lot of effort was invested in a low-value test suite.

And what of initial high coverage metrics? The verification team may very well pat themselves on the back, but could that same level of coverage have been achievable earlier on in the project, with fewer testbenches?

This chapter presents a process where coverage metrics are used to guide where to apply the next efforts in the verification process. It is of interest to project leaders and managers who need to define how to best allocate their resources and keep track of the progress of the verification project based on the requirements outlined in the verification plan (Chapter 2). The latter half of this chapter will also be of interest to verification engineers in charge of implementing functional coverage.

Developing and maintaining a good coverage model is a significant investment—worthwhile, but still significant. Despite the significance of that effort, only a few

techniques and constructs are used in the implementation of a coverage model. It is the complexity of modern designs under test that makes for a complex coverage model. A full treatment of coverage-based verification would thus have to be specific to a particular design. This chapter focuses on the design-independent elements of coverage-driven verification.

COVERAGE METRICS

Coverage metrics are measures of collected coverage data against stated or implied goals, usually expressed as percentage. Coverage data is collected in one or more databases by simulation or static tools. Coverage analysis or reporting tools compare the collected data, usually aggregated from multiple simulations or static analysis, against the goal for each coverage point. If the collected data fully covers the goal, the coverage metric is reported as 100%.

Different coverage data can be collected, as described in “Coverage Models” on page 261. The goal for a specific coverage measurement may be implied (e.g., execute every line of source code) or explicitly described (e.g., read and write from all addresses from 0x000 to 0xFFF inclusively).

The coverage metrics for multiple coverage points is usually distilled into a single overall coverage metric using a weighted average.

Coverage goals are not usually static during a verification project. An initial set of coverage goals is outlined in the verification plan. But as the implementation of testcases proceeds, those goals are updated as new corner cases and interesting conditions are identified.

Rule 6-1 — *Coverage data shall be collected as soon as it is practical.*

The incremental contribution of each and every testbench must be known. This knowledge lets the verification team focus on the high-value testbenches first and quickly identify testbenches that contribute little toward the verification objectives.

If a promising testbench turns out to contribute little toward increasing the coverage metrics, the premises behind that testbench must be questioned to identify why it is not as useful as initially thought. Is there a bug in the testbench itself? Is the feature verified in another testbench? Or are the targeted features not as significant as originally thought? By answering these questions up front, the verification efforts can be focused on the most value-add activities. Testcases should be designed to meet

their requirements—i.e. meet their coverage targets. Coverage should not be an afterthought once the testcase has been implemented.

The question then becomes: *what is practical?* "Practical" does not mean toward the end of the project, when most testcases have been written. Neither does it mean when the environment is still being developed. But it is practical to start collecting stimulus functional coverage as soon as the stimulus generators are implemented. It is also practical to implement and collect coverage data for a specific coverage point before implementing the testcase that targets it.

Rule 6-2 — *Only coverage metrics that will be looked at shall be collected.*

If a coverage metric is never looked at, if its measure of completion is not expected with baited anticipation after every regression, that coverage metric is probably irrelevant. It is important to use relevant coverage metrics. Some coverage measurements are easy to implement and can be used to produce a large number of coverage metrics. But data is not information, motion is not action. Reaching 100% coverage on an irrelevant metric may look like progress, but it is not productive work. Too many irrelevant metrics will also make the analysis task much more difficult and distract from the real verification objectives.

Rule 6-3 — *Only coverage data resulting from error-free verification tasks shall be considered.*

Coverage data is usually collected during many simulations. It may also be collected by other verification tools, such as formal verification. This data only records that a particular functional verification requirement has been exercised; coverage data does not measure a design's correctness. Coverage data must be qualified by the result of the verification task performed by the tool that collected the data. If the results were negative, the coverage data from that task must be ignored.

COVERAGE MODELS

A coverage model embodies the requirements of the functional verification process. The total stimulus and response space for a complex design is multi-dimensional and almost infinite. Thus, it is not realistic to expect to exhaustively verify a design for all possible combinations and sequences of stimuli and responses. A coverage model is a definition of the stimulus/response space subset that will demonstrate, with an acceptable degree of confidence, that the functionality of the design is correct.

For example, the solution space for a 32-bit adder is a three-dimensional space measuring 2^{32} x 2^{32} x 2 (input A x input B x carry-in). Exhaustively verifying this simple design, assuming one set of input values can be verified every nanosecond, and would require over one thousand years. But an adequate level of confidence in the correctness of an implementation can be obtained by using just a few sets of input pairs (e.g., all possible combinations of walking-ones, walking-zeroes, all-ones and all-zeroes—less than one thousand input patterns). That set of inputs is the coverage model for the combinatorial adder.

A coverage model is composed of *structural coverage* and *functional coverage* target definitions. These definitions can be further refined into sub-metrics like FSM coverage, expression coverage, cross-coverage and assertion coverage. The collection of coverage metrics and the source of those metrics is independent of the coverage model. Coverage metrics from simulation, formal analysis and structural analysis tools are combined for an overall assessment of verification progress.

Note that *combination* does not imply distillation into a single percentage number. Different coverage metrics measure different aspects of the coverage model and, like apples and oranges, cannot be arbitrarily combined. However, it is possible for a verification project to assign a relative importance to different metrics and distill a single number based on a weighted average of the individual metrics.

Structural Coverage Modeling

Structural coverage models are implicitly defined by the code used to implement the design. Structural coverage includes *line* coverage, *expression* coverage, *toggle* coverage and automatically extracted *FSM* coverage. Structural coverage also includes *assertion coverage*. Assertion coverage measures the number of vacuous, non-vacuous success and failures of assertions and the different paths and values used when evaluating them.

The collection of structural coverage metrics is automatically inserted and enabled by tools. Only their analysis requires action by the engineers. Because of the implicit and automated nature of structural coverage metrics, no additional guidelines or methodologies are required to use them.

However, structural coverage has limitations. It can only be used to measure how thoroughly an existing implementation has been exercised. It cannot identify functionality that has not yet been implemented. Neither can it determine if the intent of a testcase has been achieved.

Functional Coverage Modeling

Functional coverage modeling may appear to be a relatively recent concept, but any verification process that is based on a verification plan uses functional coverage modeling. A verification plan typically details the individual testcases that must be written to verify a particular design. The plan eventually is summarized into a table, with the testcase name in a column, the name of the verification engineer assigned to that testcase in another column and an indication of the completion status of the testcase in another. Individual engineers report their progress to a lead engineer or manager who then updates the status of each testcase in the table. The table is reviewed at regular meetings to identify if the project is on schedule, where additional resources may be needed or which testcase should be implemented next. That testcase summary table is a functional coverage model. It is simply tracked and analyzed through managerial procedures instead of automatically with tools and language constructs.

Individual testcases can be specified using functional coverage. Instead of being a mere entry in a table, functional coverage can be used to observe the simulations and determine whether a particular stimulus sequence has been applied to the design or whether the design has been through particularly interesting states. This observation would allow the progress of the verification project to be independently confirmed through an automated coverage tool. It would also eliminate the risks of a bug in a testbench that, unbeknownst to the author, prevents some portions of the test from being executed on the design. Without a failure to indicate a problem, such a testbench would be assumed—wrongly—to be complete and to cover its intended objectives.

When the functional coverage of individual tests is measured automatically, it becomes irrelevant how that test happens to have been covered. It may be implemented in a directed testcase, hit by pure chance by a random simulation or proven to be correct by a formal analysis. In a coverage-driven verification process, the strategy becomes how to use the best implementation vehicle to hit the most coverage points with the least effort. If individual directed testbenches are required to hit individual coverage points, it will be necessary to write as many testbenches as there are coverage points. By adding randomness and simulating the same testbench with several different seeds, the same testbench may be able to hit multiple coverage points, effectively replacing multiple directed testbenches.

Randomness can be pushed so far that it becomes impossible to know, a priori, what many aspects of a simulation will do. That level of randomness is not very useful unless the simulation also records functional coverage metrics. With functional coverage metrics, it is possible to know, after the fact, which coverage points were hit

by the random simulation. With enough degrees of freedom in the random generators and multiple simulations with different seeds, it is possible to replace several directed testbenches with one random testbench. Additional coverage points can be targeted by adding or relaxing constraints on the random testbench, creating new testbenches with little additional effort.

Like structural coverage, functional coverage has limitations. It can only demonstrate that some interesting condition has been reached, not that it was reached correctly or that the corresponding responses were correct. It does not imply completeness: If a functional coverage models only 70 percent of the DUT's functionality, then reaching 100 percent functional coverage will only verify that 70 percent. Functional coverage is an expression of the verification requirements and intent. Therefore, it cannot be automatically derived or implemented from the DUT or testbench source code. It must be manually specified and coded, an often tedious task.

There is no way to ensure that a functional coverage model is complete because it is only a reflection of the verification plan, and there is no automated way to ensure that the verification plan itself is complete. A functional coverage model should be subjected to the same type of development and review process as the verification plan to ensure its completeness. It will also evolve throughout the duration of the verification project, as new functionality is added or new corner cases are identified.

Recommendation 6-4 — *Directed tests should include functional coverage.*

Functional coverage is implicit in directed tests. Because the test is directed, the verification requirement targeted by the test is known. Assuming that the directed test does what it is intended to do and the simulation completed without errors, it can be assumed that the functional coverage point associated with the targeted verification requirement is covered.

The keyword here is "assume." What if there is a bug in the testbench? What if there is an architectural change in the design that modifies how the verification requirement can be met? Tests should confirm that they have met their verification requirements.

Recommendation 6-5 — *Functional coverage points should be implemented before the testcase that targets them.*

It is possible that an existing random or directed testcase already exercises the functionality that is the objective of the next testcase to be written. By implementing functional coverage first then waiting for the coverage results from the next regression, the coverage points will be automatically filled if they are—accidentally—exercised. If they are, then there is no need to implement the testcase.

If some (or all) coverage points are not filled, then the testcase should be implemented in the most efficient manner to target the remaining uncovered coverage point. This may be a directed testcase, a new constrained-random testcase, or a formal analysis.

Functional Coverage Analysis

Functional coverage analysis is the process of identifying uncovered areas in the coverage model, then identifying the next functional verification requirement that should be targeted.

Recommendation 6-6 — *Functional coverage data should be designed to facilitate analysis.*

The initial reaction of first-time coverage users is to worry about the runtime cost of collecting functional coverage data. But this cost pales in comparison to the cost of analyzing the collected data. The collected coverage data must be optimized to ease its interpretation first. Optimization for runtime performance should come second.

The best way to optimize for interpretation is to understand the difference between *data* and *information*. Data is a collection of raw numbers; information is the knowledge extracted from them. Instead of covering raw data, information should be covered.

For example, the coverage of a 1K-deep FIFO could collect read and write pointer values. But this approach creates one million different coverage points and makes it difficult to relate the coverage space to the verification requirements.

A more useful coverage would be the occupancy level of the FIFO. Although this makes the correlation of the coverage space to requirements much easier, it still yields one thousands separate coverage points. The best coverage is whether the FIFO was ever full or empty. This data reduces the number of coverage points to two and creates the simplest coverage space that can be easily related to the verification requirements. Should the FIFO Full coverage point be empty, it clearly implies that no simulation every filled the FIFO. That type of information is much more difficult to deduce from the more complex coverage spaces.

A complete functional coverage of a FIFO would eventually include conditions such as read pointer ahead/behind the write pointer, read/write when full/empty, simultaneous read and write and full/empty with pointers at minimum/maximum values.

Coverage Grading

A common worry when using coverage metrics in tandem with random stimulus is the possibility of witnessing a significant drop in the coverage rating simply because different seeds were used.

To avoid this problem and help optimize the simulation cycles, coverage analysis tools can also grade the various coverage data sources. The grading helps determine which simulation or analysis contributes the most toward the current coverage rating. Grading can be based on absolute contribution to the coverage rating or rate of contribution over time. These simulations and analyses, together with their initial seed value, are collected to create the regression suite. By running the most efficient simulations and analyses first, the same level of coverage rating can be reliably obtained in less time.

FUNCTIONAL COVERAGE IMPLEMENTATION

Metrics in a structural coverage model can be automatically generated and collected. For example, to obtain a rating of the line execution coverage, all that is required is the specification of a command-line option. No additional work is required by the user— and no excuses can be presented against using it.

But a functional coverage model must be manually specified and captured. Functional coverage—which models the functional verification requirements—captures the intent of the verification effort. Because intent cannot be automatically derived from the implementation, functional coverage must be independently captured.

Two constructs are available in SystemVerilog to specify functional coverage: *coverage groups* and *coverage properties*.

Rule 6-7 — *Coverage groups shall be used when covering data in the verification environment.*

Coverage properties use concurrent temporal expressions to define cover points. In SystemVerilog, temporal expressions support references to static variables—variables and nets in module and interface—only. Because the verification environment is constructed using `classes`, which are dynamic constructs, concurrent temporal expressions cannot be used.

Recommendation 6-8 — *Coverage properties should be used to specify implementation-specific functional verification requirements.*

As described by Recommendation 2-10, some functional verification requirements are dictated by the implementation chosen by the designer and are not apparent in the functional specification of the design. These requirements create corner cases that only the designer is aware of. These requirements should be specified in the RTL code itself by the designer through coverage properties.

Example 6-1. Implementation-Specific Coverage Property

```
cover_stack_hi_water_chk :
   cover property( @(posedge sva_checker_clk)
      not_resetting &&
       $rose(sva_v_stack_ptr == hi_water_mark));
```

Recommendation 6-9 — *Coverage properties should be used to specify physical interface compliance requirements.*

A design must be compliant with the specification of the physical interfaces present on its periphery. To ensure compliance, the design must be verified to operate correctly under various conditions of the physical interface. These conditions, often called *protocol compliance statements*, should be specified using coverage properties.

Recommendation 6-10 —*Coverage groups should be used if the sampled data must be mapped into different coverage points.*

Coverage properties can specify only a single coverage point. If a sampled value is to be mapped into different coverage points, a coverage group is better suited with its `bins` option in the `coverpoint` element.

For example, sampling the value of an address bus may need to be mapped into different coverage points, one per target device in the address space. This mapping can be more easily described using a single coverage group than multiple coverage properties.

Example 6-2. Mapping Sampled Data into Different Coverage Bins

```
frame_len: coverpoint frame_len {
   bins max_fl_less_4 = {cfg.max_frame_len - 4};
   bins max_fl_less_1 = {cfg.max_frame_len - 1};
   bins max_fl        = {cfg.max_frame_len};
   bins max_fl_plus_1 = {cfg.max_frame_len + 1};
   bins max_fl_plus_4 = {cfg.max_frame_len + 4};
   bins max_size      = {65535};}
```

Recommendation 6-11 —*Coverage groups should be used if the sampled data must be crossed-covered with other data.*

Coverage properties can specify only a single coverage point. If two sampled values must be combined to form a two-dimensional space in the coverage model, it would require an exponential number of coverage properties. The `cross` element in a coverage group can easily describe multi-dimensional combinations of sampled values.

Example 6-3. Crossing Coverage Elements

```
covergroup frame_coverage;
   ...
   cross direction, hugen, max_fl, frame_len;
endgroup
```

Coverage Groups

The `covergroup` statement specifies functional coverage points by sampling variables and expressions. A coverage group specifies *what* is being sampled, *when* it is sampled and *how many* samples are required to consider a coverage point "filled."

Suggestion 6-12 — *The data sampled by the coverage point need not be the data sampled in the testbench.*

The data sampled by the coverage points is the data that will be recorded in the coverage database and will be available for analysis. As specified in Recommendation 6-6, the collected coverage data should be designed to ease analysis, not simplify the implementation of the coverage group. Therefore, the data sampled by the coverage group may have to be a more informative or transformed version of the data sampled in the verification environment.

It is tempting to use a convenient clock signal to sample in-scope variables, but it may not provide the type of information that is easy to analyze. It may be necessary to introduce additional behavioral code in the verification environment to interpret the available data into useful coverage data and manually trigger the sampling of the coverage group using the `sample()` method.

Example 6-4. Computing Information to be Covered

```
this.is_tx     = 1;
this.frame_len = fr.byte_size();
this.tx_size_cvr.sample();
```

Recommendation 6-13 —*Sampled values should be mapped to a manageable maximum number of explicitly named* `bins`.

The mapping process interprets the sampled values into different coverage points. If no `bins` are specified, the automatically generated bins will contain only one value and will not have a meaningful name. For a 32-bit value, this represents over four billion bins, which is a huge coverage space and an unachievable coverage goal. A large number of bins makes it impossible for a coverage analysis tool to enumerate all uncovered coverage points. And those are the coverage points that are the most interesting!. Having a large number of bins also creates explosions when cross-coverage is used.

Limiting the number bins to 10 for example, will limit two-dimensional cross-covers to 100 coverage points. By reducing the number of bins—and thus coverage points, the coverage space is reduced and makes reaching a coverage goal of 100% quite achievable. There must be an individual bin for each verification requirement. If two requirements are mapped to the same bin—or coverage point—it will be impossible to determine if one of the two requirement has not been met.

Example 6-5. Minimizing the Number of Bins for a 16-bit Sample Value

```
frame_len: coverpoint frame_len {
   bins max_fl_less_4 = {cfg.max_frame_len - 4};
   bins max_fl_less_1 = {cfg.max_frame_len - 1};
   bins max_fl        = {cfg.max_frame_len};
   bins max_fl_plus_1 = {cfg.max_frame_len + 1};
   bins max_fl_plus_4 = {cfg.max_frame_len + 4};
   bins max_size      = {65535};
}
```

Rule 6-14 — *Stimulus coverage shall be sampled after submission to the design.*

Not all sequences of transactions that are generated will be applied, as-is, to the DUT. In a transactor execution, the scenario may have filtered out some transactions from the sequence, or the scenario may have injected errors that caused some transactions to be ignored or rejected by the DUT.

Scenarios are composed of high-level transactions that are composed of command-level transactions. As transactors execute transactions, they can relate low-level transactions back to the higher-level transaction that caused it to provide some context for the low-level transaction. As high-level transactions and scenarios are generated, they must be buffered until they have been completely implemented by the lowest-layer of the environment.

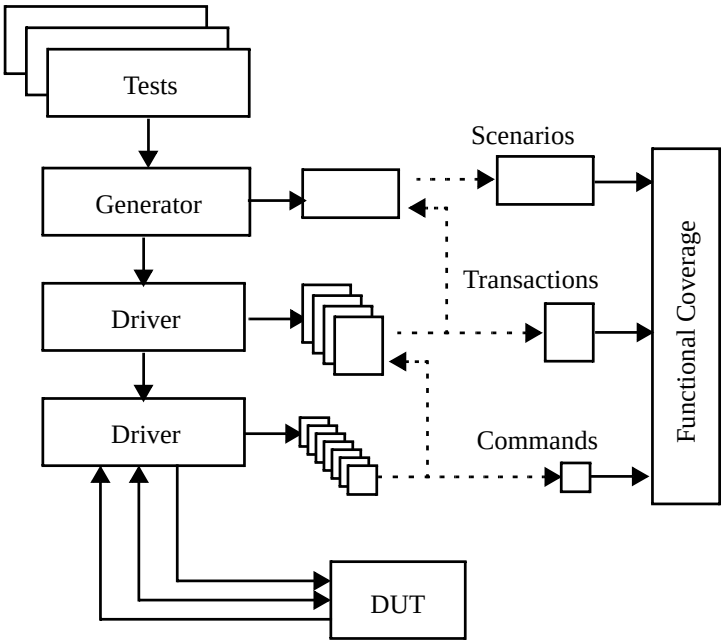


Figure 6-1. Collecting Stimulus Coverage

As shown in Figure 6-1, low-level transactions are covered first. Using the context information in the low-level transaction, the coverage model can determine if it is the (successful) termination of the higher-level transaction or scenario it is part of. If it is, the higher-level transaction is also recorded in the coverage model. If any low-level transaction is missing or indicates an improper implementation of a scenario or high-level transaction, the scenario or high-level transaction is dropped and never included in the coverage model.

Recommendation 6-15 —*Stimulus coverage should be sampled via a passive transactor stack.*

Figure 6-2 shows two alternatives for collecting stimulus functional coverage in a verification environment. In Figure 6-2(a), the functional coverage is collected in the generation and driver stack. In Figure 6-2(b), it is collected in a passive transactor stack. The latter has the additional advantage of being portable to a different verification environment that uses a different stimulus generation structure (such as directed testcases) or to a system-level environment where the stimulus is generated by another design block.

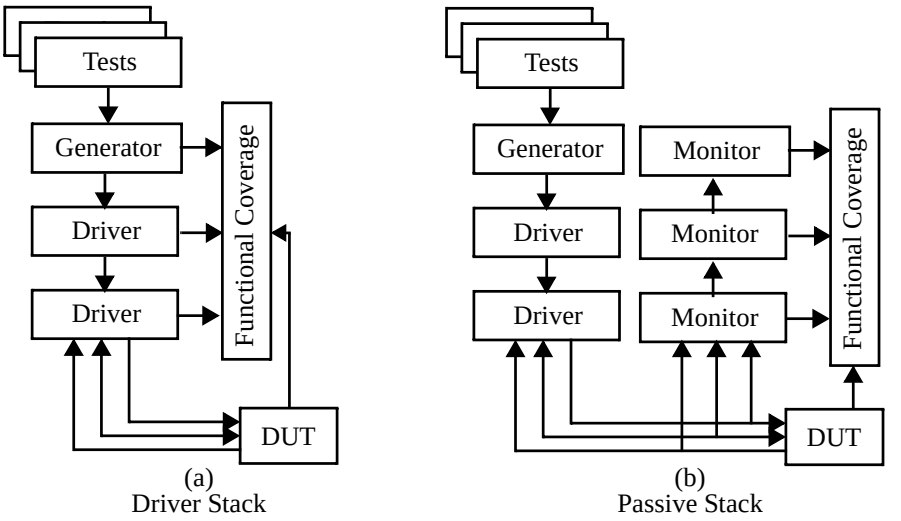


Figure 6-2. Collecting Stimulus Coverage

Using a passive transactor stack to collect functional coverage directly from the DUT interface ensures that only what has been observed by the DUT will be recorded in the functional coverage model. This implementation approach satisfies Rule 6-14.

Suggestion 6-16 — *DUT state coverage points can be implemented using coverage groups instantiated in the self-checking structure.*

Some states of the DUT can be inferred from the operations and transformations that are necessary to verify the correctness of the DUT response. If no errors were detected, it can be implied that the DUT and self-checking structure have performed identical operations or transformations. By sampling data in the self-checking structure that is representative of the relevant DUT state, state functional coverage can be implemented in the self-checking structure.

For example, not adding a packet to the scoreboard because it is invalid can be sampled in the state functional coverage to infer that the DUT has rejected an invalid packet. If that packet is indeed never observed on the output (i.e., no errors are detected), the inferred state functional coverage is valid.

Because of its DUT-specific nature of a self-checking structure, it will always be developed by the verification team and the source code will always be available.

Since the source code is available, it can be modified to insert the DUT state functional coverage model.

Example 6-6. Coverage Group in Self-Checking Structure

```
class scoreboard;
   ...
   covergroup frame_coverage;
      ...
   endgroup: frame_coverage
   ...
endclass: scoreboard
```

Recommendation 6-17 —*Coverage groups should not be added to* `vmm_data` *class extensions.*

The `vmm_data` base class is designed to help model data and transaction descriptors. Although it may be seem natural to put the data and transaction-related coverage groups in those same classes, it creates several limitations.

There are hundreds of thousands of data and transaction descriptors created and garbage collected in the course of an average simulation. A corresponding number of coverage group instances will therefore need to be created and tracked. Each coverage group will contain coverage data for a single data or transaction instance only. Individual coverage metrics will be meaningless and only cumulative coverage will be useful. It would be more efficient to collect hundreds of thousands of coverage samples in a single instance of a coverage group.

If a verification environment has multiple streams of the same data or transaction descriptors, it will be necessary to cross-cover all sampled data with a stream identifier to differentiate the coverage metrics on a per-stream basis. If each stream has its own set of coverage group instances, coverage metrics for each stream will be individually collected for the thousands of data or transaction descriptor in each stream, without having to use cross-coverage. Cumulative coverage can also be used to report coverage metrics regardless of the stream where the data was collected.

Recommendation 6-18 —*Functional coverage should be associated with testbench components with a static lifetime.*

To avoid creating a large number of coverage group instances with few coverage data in them, they should be associated with objects with transactors, monitors, generators, the self-checking structure or the design under verification. They are all created at the beginning and live until the end. By associating functional coverage groups with these verification environment components, data and transaction descriptors can be

sampled as they flow through them. It will thus be possible to measure coverage for individual transactor instances and cumulative coverage for all instances of a particular transactor.

Recommendation 6-19 —*Coverage groups should be encapsulated in coverage objects.*

Instantiating coverage groups in coverage objects will allow the encapsulation of the transformation of the data as sampled on the verification environment or the DUT into a form that can be sampled by the functional group themselves to fill coverage points.

Rule 6-20 — *The data sampling interface of the coverage class shall be designed to match the verification environment.*

Inserting functional coverage into a verification environment must be as unobtrusive as possible and require the minimum number of extensions. Matching the data sampling interface of the coverage object to the verification environment makes the integration of the coverage model in the environment that must easier.

As illustrated in Figure 6-3, any discrepancies or gaps between the data sampling interface and the sampled values in the functional coverage group are bridged by transformations and computations inside the coverage class itself.

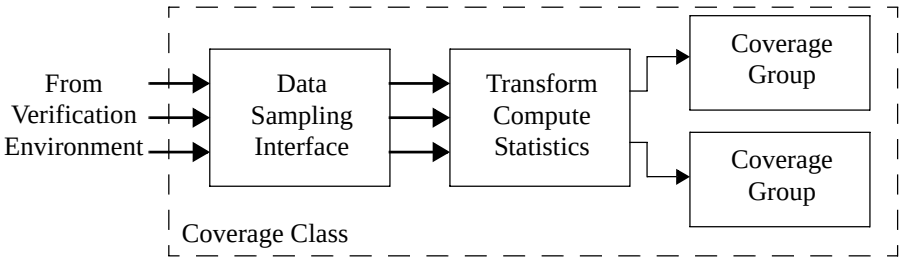


Figure 6-3. Structure of Functional Coverage Class

The data sampling interface should be designed to match the available data as reported by the various transactors via callback methods or in public properties.

Rule 6-21 — *The coverage class shall reconcile sampling domains when crossing variables sampled in different sampling domains.*

The sampling mechanism built into the functional coverage groups only allows for a single sampling event to cause the sampling of all sampled variables. Because cross-

coverage is not allowed across coverage groups, all variables to be cross-covered must be sampled by the same coverage group—hence, the same sampling event. With variables in different sampling domains that are valid at different (and potentially asynchronous) points in time, this structure may yield false coverage points.

Cross-covering data across sampling domains requires resolution of the sampling domain differences by the user. This resolution can be done using arbitrary SystemVerilog code in a coverage object to concurrently sample, each in their own domain, all required variables. Then when an appropriate window is identified, sample the intermediate values into a coverage group to be crossed. This approach is illustrated in Figure 6-4.

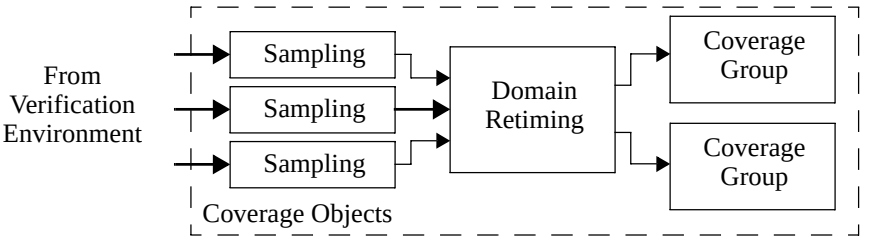


Figure 6-4. Cross-Coverage Across Sampling Domains

Recommendation 6-22 —*The coverage weight of* `coverpoint` *involved in a cross-coverage should be set to zero.*

If the cross-coverage goal is met, it usually implies that the coverage goal of the individual samples in the cross-coverage have been met. Setting their coverage weight to 0 prevents the overall coverage score of the coverage group from being artificially raised. The score would have been raised by having some (or all of the) components of the cross-coverage fully covered but not the cross-coverage itself. Because the cross-coverage is more representative of the true coverage goal, it alone should contribute to the coverage group score.

Example 6-7. Setting Weight of Coverpoint to 0

```
covergroup frame_coverage;
   direction: coverpoint is_tx {
      bins Tx = {1};
      bins Rx = {0};
      option.weight = 0;
   }
   ...
```

```
      cross direction, ...;
endgroup: frame_coverage
```

Recommendation 6-23 —*The coverage weight of a* `covergroup` *should be set to the number of* `coverpoints` *and* `cross-points` *with a non-zero coverage weight.*

By default, the coverage weight of a coverage group is the same whether the coverage group contains a single sample or dozens of large cross-coverages. The latter is much more difficult to fill than the former and thus, should carry a greater weight in the overall coverage rating. This guideline weighs coverage groups proportionally to the number of relevant coverage samples and crosses it contains.

Example 6-8. Setting Weight of Covergroup

```
covergroup frame_coverage;
   direction: coverpoint is_tx {
      bins Tx = ...;
      bins Rx = ...;
      option.weight = 0;
   }
   hugen: coverpoint cfg.huge_enable {
      option.weight = 0;
   }
   max_fl: coverpoint cfg.max_frame_len {
      bins fl_1500 = ...;
      bins fl_1518 = ...;
      bins fl_0600 = ...;
      option.weight = 0;
   }
   frame_len: coverpoint frame_len {
      bins max_fl_less_4 = ...;
      bins max_fl_less_1 = ...;
      bins max_fl        = ...;
      bins max_fl_plus_1 = ...;
      bins max_fl_plus_4 = ...;
      bins max_size      = ...;
      option.weight = 0;
   }
   cross direction, hugen, max_fl, frame_len ...
   option.weight = 2 * 3 * 6;
endgroup: frame_coverage
```

Rule 6-24 — *The sampling frequency of a coverage group shall be minimized.*

Every time the sampling event of a coverage group occurs, a potentially large amount of data is sampled, expressions are computed and assigned to state bins and a

database is updated. This activity consumes simulation bandwidth and reduces performance.

Although tools are optimized for performance, they should not be overburdened. Data for functional coverage should be sampled as little as possible: Repeatedly sampling the same value reduces performance while providing no new information. For example, avoid sampling data using a clock. Instead, use another event that is more indicative of a potential change in value of the sampled data.

Recommendation 6-25 —*The* `comment` *option in* `covergroup`, `coverpoint` *and* `cross` *should be used to document the corresponding verification requirements.*

Coverage points specify the functional coverage requirements of the verification project. Thus, they should be cross-referenced to the requirement they implement to ensure that all requirements are covered by the coverage model.

If a URL is specified in the `comment` string, coverage analysis tools should be able to hyperlink the coverage item to the documentation describing the verification requirement implemented by the coverage item.

Example 6-9. Documenting Coverage Requirements

```
covergroup frame_coverage;
   ...
   cross direction, hugen, max_fl, frame_len {
      option.comment = "Coverage for Example 2-5";
   }
   ...
endgroup: frame_coverage
```

Coverage Properties

Properties can be functionally covered to determine if they have had the opportunity to fire, and if they completed successfully during a simulation.

The same mechanism can be used to implement coverage points. A coverage point is described using a concurrent temporal expression—or `property`. The temporal expression is not designed to perform checking and report an error if some conditions are seen or fail to materialize. It is designed to simply report occurrence of an interesting condition in the design. By covering these expressions, they can form additional coverage points in the coverage space.

Guidelines described in Chapter 3 can be used to implement coverage properties. Furthermore, following the guidelines described in Chapter 7 may enable formal tools to cover coverage points specified using properties.

FEEDBACK MECHANISMS

In a coverage-driven verification process, the analysis of the collected coverage data is used to identify where to focus the next verification efforts. Thus, a feedback mechanism is required to translate uncovered areas in the coverage model into additional stimulus or analysis.

Recommendation 6-26 —*A manual coverage feedback mechanism should be used.*

There are mechanisms in SystemVerilog to dynamically query the coverage rating of different coverage points. The information can then be used to alter, at run time, state variables in constraints on the stimulus. A simulation could therefore automatically focus on the uncovered areas of the coverage space and reduce the likelihood that the same coverage points will be repeatedly hit.

That's the theory. In practice, things are a little different.

First, this focusing on uncovered areas can only work on input coverage, where there can be a direct correlation between the coverage rating of particular coverage points and state variables in stimulus constraints. This relationship quickly becomes tenuous at best for internal coverage points or output coverage.

Furthermore, this automated feedback mechanism is not trivial to implement. It must be coded and debugged to ensure that the stimulus converges toward greater coverage. The objective is not to minimize the number of testbenches, but to minimize the effort required to reach the desired coverage goals. It is likely to be easier to reach with more trivial testbenches than a few complex ones.

By intellectually identifying how the stimulus must be constrained or directed, it will be easy to specify that new stimulus in an additional test that specifically targets holes in the coverage space. These tests should be simple to write and debug—as long the the guidelines presented in this book are followed. They are run for a few seed values and added to the regression suite. With a few iterations, a particular set of related coverage points should be covered in less time than it would take to automate the process to do the same in a single testbench.

There are exceptions to this guideline. In some circumstances, it may be worthwhile to invest the time and effort to close the coverage loop automatically. For example, if a testbench is going to be reused in different versions of the simulator and constraint solver, it may generate different stimulus for the same seed values. Also, this testbench may be a useful mechanism for demonstrating the operation of the design to a third party.

At the time of writing, researchers are studying techniques for automatically closing the coverage feedback loop without additional requirements from the user.

Recommendation 6-27 —*More simulations with different seeds should be run if the input coverage is low and few simulations have been run.*

If the coverage metrics report that only 8 out of 10 opcodes have been observed in the instruction decoder, and only three simulations have been run so far, run more simulations.

Recommendation 6-28 —*Quality of distribution in the generator should be improved if the input coverage is low and many simulations have been run.*

If the coverage metrics report that only 8 out of 10 opcodes have been observed in the instruction decoder, and 25 simulations have been run so far, there is a problem with the distribution in the instruction generator.

Example 6-10. Matching Generation Distribution with Coverage Requirements

```
covergroup frame_coverage;
   ...
   max_fl: coverpoint cfg.max_frame_len {
      bins fl_1500 = {1500};
      bins fl_1518 = {1518};
      bins fl_0600 = {'h0600};
      weight=0;
   }
   ...
endcover: frame_coverage
...
class test_cfg;
   ...
   rand bit [15:0] max_frame_len;
   ...
   constraint coverage_driven {
      max_frame_len inside {1500, 1518, 'h0600};
   }
```

```
    ...
endclass: test_cfg
```

Recommendation 6-29 —*A new test should be created if constraints need to be modified or new scenarios defined.*

If a test did not hit certain coverage points it was designed to hit, fix the test. But to hit additional coverage points, a new test should be created. This approach will allow the existing test to continue working in the regression suite and maintain the existing coverage level.

If the guidelines outlined in this book are followed, the amount of code required to write a new test should be minimal.

Recommendation 6-30 —*A directed test should be used if a scenario is too complex to define or unlikely to happen randomly.*

If it is unlikely that the required stimulus will be randomly generated, specify a directed stimulus sequence. For example, verifying that all status bits will generate maskable interrupts would require that they be randomly masked and unmasked, then that the design randomly hits each exception condition, then they be randomly masked and unmasked once the corresponding status bit is set. This sequence is unlikely to occur—or very difficult to describe—in a random test. But it is a simple directed test, especially if the status bits can be forced.

However, it need not be a 100% directed testcase. It can be a directed stimulus sequence randomly injected within a stream of other random sequences. This approach may highlight problems that may not be apparent if the directed stimulus is always applied from the reset state.

Recommendation 6-31 —*Formal technology should be used if the problem size fits the capacity of the tool and a coverage property is used.*

It is very difficult to hit a coverage point embedded deep in the design; it may be easier to use formal technology to meet this functional verification requirement. The next chapter will detail how formal technology can be used to its best effects.

SUMMARY

This chapter defined the concept of coverage metrics and how they form a coverage model. The coverage model is an embodiment of the verification requirements. A coverage model is composed of structural coverage metrics and functional coverage metrics. The structural metrics can be automatically collected and measured. Because functional coverage metrics specify verification intent, they must be implemented manually.

Next, guidelines were specified on how to best implement functional coverage. Functional coverage can be implemented using coverage groups or coverage properties. When collecting coverage data, it is important to ensure that the data is in the form that will be observed by the DUT. It is also important to reconcile data from different clock domains.

Coverage metrics must be designed to facilitate analysis of the holes. From a coverage report, it should be easy to identify which testcases need to be created next. The feedback from functional coverage metrics to testcases can be automated. But in most cases, an intellectual feedback process using different closure mechanisms based on the targeted holes is more efficient.

CHAPTER 7 ASSERTIONS FOR FORMAL TOOLS

The SystemVerilog assertions language has well defined semantics for both simulation and formal tools, in particular, *model checkers*. While it is possible to simulate almost any assertion given enough memory and time, the capacity constraints on formal tools are much tighter. Furthermore, the latter usually require that the model structure including the assertions be statically determined. Similar constraints apply in emulation.

Therefore, when writing assertions that should be used in formal/emulation and in simulation, the guidelines expressed in this chapter should be considered in addition to those in Chapter 3. In other words, the decision to use (even potentially) formal tools or emulation influences the assertion style and it should be made early in the development process.

Why and when should formal tools be used if it means sacrificing some freedom of expression on the assertions side? It has been shown that formal tools are very effective in finding corner-case bugs on complex control-dominated design blocks, such as arbiters, bus protocol controllers, instruction schedulers, pipeline controls, and so on. Another useful application is to check relatively simple structural properties that can be automatically extracted from the design, e.g., bus driver mutual exclusion, parallel and full case check and clock domain crossing. If the design contains such structures, it is advisable to use formal tools to identify and fix any bugs they can find before committing the design to silicon. That is, use formal tools after most of the simpler problems have been eliminated using simulation-based tests.

This chapter provides a brief introduction into model checking and the possible uses of assertions with formal tools. The main part of the chapter provides guidelines for

the style to be used when writing assertions for such tools. These guidelines also apply to using assertions in emulation because emulation has similar resource constraints and synthesizability requirements to formal tools. The chapter should thus be of interest to anyone who plans to use SystemVerilog assertions with formal tools, emulation or hardware prototyping.

MODEL CHECKING AND ASSERTIONS

The formal tools used to verify properties specified using a temporal language have been usually called *model checkers* or *property checkers*. They verify that the DUT is a model of the temporal expression of the assertion, i.e., the behavior of the model satisfies the property.

Model checking has evolved considerably over time. At the time of writing, model checking tools often involves many different verification engines. The engines may implement binary-decision-diagram-based (BDD-based) reachability algorithms, satisfiability (SAT) algorithms, automatic test pattern generation (ATPG), symbolic simulation, regular value-based simulation, and so on. The engines usually operate with efficient model reduction and abstraction mechanisms, all carefully orchestrated during their execution to reap the maximum benefit in performance and the ability to reach a definitive conclusion: the tool finds a violation of the property or it proves it correct on the DUT.

Unfortunately, all the powerful formal verification algorithms embedded in the engines solve problems that have non-deterministic polynomial complete (NP-complete) complexity or worse. Hence, they must rely on heuristics. It then follows that, due to the complexity of the DUT or the properties, the verification process may run out of available time or memory: The result is inconclusive. The user must then simplify (break up) the property or reduce the DUT to something more manageable.

Formal tools accept a set of temporal properties as statements about the DUT behavior. Usually, they also require a model of the environment of the DUT. This model of the environment can also be described using temporal properties. In most cases, all variables are expanded to the bit level. Very few commercial tools (if any) can handle word-level variables and operations.

The tools may then carry out the following tasks:

1. Given a set of properties that represent assumptions on the behavior of the environment, prove that the DUT satisfies the assertions on its behavior.

The main goal is to find any design errors that would falsify the assertion. The tool may use algorithms that favor bug hunting over proofs. When a failure is detected, the tool generates a test sequence that drives the design to the failure state (the counter-example). This trace can be used with a simulator to debug the design.

2. Given a coverage goal specified as a `cover property` statement or a particular signal state of the design, the tool generates a stimulus sequence that drives the design to reach that goal.

   A set of assumptions on the environment is still needed as in (1) above to avoid generating tests that are outside the expected behavior of the environment of the design.

3. Given a set of state variables in the DUT, the tool determines a lower bound on the set of states that cannot ever be reached.

   A set of assumptions on the environment is useful to make the bound tighter. This set of unreachable states is useful in determining the maximum obtainable state coverage by identifying and removing unreachable states from the coverage model. Also, some of the unreachable states may actually represent design errors.

Formal tools generally require that the model of the DUT and the expressions, auxiliary variable types and assignments used with and in assertions be synthesizable. It is generally understood that synthesizable assertions means that the code used to implement the assertions satisfy similar syntactic restrictions as RTL code. In particular, only bounded data types and statically allocated variables can be used and no assignments or comparisons with the values `X` or `Z` can be made.

Except for local variables, SystemVerilog assertions are based on extended regular expressions and thus can be relatively easily compiled into equivalent RTL logic. When local variables are declared as part of `sequence` or `property` definitions, they are allocated dynamically at run time, as needed in individual evaluation attempts and threads. This language feature increases considerably the modeling power and ease of use of the assertion language. However, it is possible to write properties that may require an unknown (possibly unbounded) amount of dynamically allocated resources, which cannot be synthesized

With a sufficient amount of resources, it is possible to implement local variable allocation mechanisms in hardware. The problem is that such properties may easily exceed the capacity of the formal tools. Therefore, it is far better to recognize such limits up front and provide guidelines regarding the style to be used for synthesizable assertions. The style then guarantees that the assertion can be converted into an efficient synthesizable RTL model.

Even if synthesizable, not all types of designs and assertions can be accepted by formal tools because of capacity limitations. These limitations are specific to a particular tool and the type of proof engines used within that tool. Even when the type of engine is known (e.g., BDD-based reachability), it is impossible to determine a priori how the tool will behave on a particular design or property. There are a few general guidelines that may help to avoid capacity problems. In addition to the guidelines specified in Chapter 3, the rest of this chapter contains three kinds of guidelines: general guidelines that should be followed in all cases, style guidelines showing how certain behaviors can be modeled without using local variables, and guidelines that indicate what kind of structures can be used with local variables and yet remain synthesizable.

In general, assertions can be proven or failed faster with fewer assumptions. While trying to minimize the number of assumptions, it is easy to over- or underconstrain the inputs. Overconstraining[1] inputs leads to false positives whereas underconstraining[2] inputs leads to false negatives. Finding the correct combination of assumptions that neither over- or underconstrain the inputs is often a difficult task. It is therefore important to have access to high-quality assertion-based verification IP (AIP), as discussed in Chapter 3.

Some formal tools—like Synopsys' Magellan—may use constrained-random simulation as one of the verification engines. Random stimulus engines are mainly employed in bug-hunting mode, since a proof is not possible in that case. Without any specific user-written testbench, the simulation stimulus is randomly generated within the constraints imposed by the assumption properties on the environment. In addition to false positives and negatives, improperly constrained inputs in a random generation engine may lead to dead end conditions, forcing the analysis to a halt. The tool may also contain an analysis engine for determining and eliminating dead-end states in the set of assumptions. It can provide a list of situations where a dead end occurs and suggest additional constraints that would avoid reaching the dead end.

Both under- and overconstraint analysis and resolution are difficult problems and can be quite time consuming. These tasks also require a good knowledge of the expected behavior of the environment. The guidelines in this chapter may help in the formulation and debugging of both assertions and assumptions for use with hybrid-formal tools.

1. Overconstraining means that the space of the possible input sequences is smaller than what the DUT is designed for.
2. Underconstraining means that the space of possible input sequences is larger than what the DUT is designed for.

Recommendation 7-1 — *The simulation testbench should be used to validate the assumptions on the DUT environment.*

If a simulation testbench exists for the DUT, it is useful to verify that the properties on the DUT environment do not fail when running as assertions in simulation. A failure would indicate either a problem in the testbench or that the assumption is not correct. Both the testbench and the assumptions can be jointly debugged.

Rule 7-2 — *Non-synthesizable assertions shall be under the control of the macro* `VMM_FORMAL`.

As mentioned in Rule 3-58 on page 80, assertions that check for the presence of `'x` or `'z` using case equality (= = =) are not synthesizable and may not be used with formal tools. Therefore, the inclusion of these assertions shall be controlled by a macro.

Recommendation 7-3 — *Properties that verify end-to-end behavior of the DUT at the transaction level should be avoided.*

At the transaction level, the behavior is often expressed in terms of information exchanges using (potentially long) blocks of data or packets. The events that determine the validity of the generated or received data are often on a different time scale than the system clock of the design. Also, the data decoding, checking and routing may involve complex algorithms. End-to-end properties may also involve most of the behavior of the block so that there may be little model reduction possible. If, in addition, the size and sequential complexity of the block is high, it is then likely that the formal tool will not be able to complete the proof. However, if the tool is organized for bug hunting using a mix of simulation and formal engine, it may still be useful to probe the design in this mode using such a property.

Recommendation 7-4 — *Proving properties that involve multiple complex design blocks should be avoided.*

Such properties may only allow minimal model reductions and thus may cause the problem to exceed the capacity of the tool. It is better to provide assertions that characterize the behavior of each block on each interface and use them as assertions/ assumptions to verify properties of each block independently.

Recommendation 7-5 — *The DUT should be architected as an interconnection of blocks that have well-defined interfaces with their neighbors.*

Such an architecture simplifies proving properties of each block independently as recommended by Recommendation 7-4.

Recommendation 7-6 — *Properties should be broken into a series of smaller ones, such that, collectively, they are equivalent to or imply the original property.*

As mentioned in Chapter 3, smaller and simpler properties facilitate understanding and debugging. Also, they provide finer control when adding or removing assumptions to prove properties. It is usually more effective to prove assertions with the smallest possible number of assumptions on the environment.

Example 7-1. Breaking a Property into Smaller Ones that Imply the Original

```
// If v occurs then w must be asserted for one cycle
// followed by a z within 0 to 4 cycles
property p;
  @(posedge clk)
    v |-> w ##1 !w ##[0:4] z;
endproperty : p

// Break up into 3 properties
  property p1;
    @(posedge clk)
    v |-> w; // if v then w
  endproperty : p1
  a_p1: assert property (p1);

  property p2;
    @(posedge clk)
    w |-> ##1 !w;
  endproperty : p2
  a_p2: assert property (p2);

  property p3;
    @(posedge clk)
    w |-> ##[1:5] z;
  endproperty : p3
  a_p3: assert property (p3);

  property p_auxiliary;
    @(posedge clk)
    w |-> v;
  endproperty : p_auxiliary
  a_p_auxiliary: assert property(p_auxiliary);
```

In Example 7-1, properties *p1*, *p2* and *p3* imply collectively *p*. However, if *w* can also occur without a *v*, then *p2* and *p3* could trigger without a *v*, that is, *a_p2* or *a_p3* could fail while *a_p* would not. If a *w* should not occur without a *v* then add the property *p_auxiliary* and the associated assertion *a_p_auxiliary* to verify that assumption.

Recommendation 7-7 — *Proving the correctness of arithmetic operations using a high-level description should be avoided.*

Property-checking algorithms used in commercial tools are generally weak on proving arithmetic operations because they try to compare the implementation against a functional specification. This weakness is particularly apparent in the case of multiplication where the internal data structures of the tool tend to grow exponentially in size with the number of bits in the operands. The reader should consult the literature on formal verification for techniques how to deal with arithmetic operations on large words.

Example 7-2. Avoid Property on an Arithmetic Operation

```
reg [31:0] a, b, result;
reg , reset_n, clk;
// a multiplier, result is available
// at the next clock cycle
multiplier mult_inst(clk, reset_n, a, b, result);
bad_property: assert property (
    @(posedge clk) disable iff(!reset_n)
                  result == $past(a * b) );
```

Recommendation 7-8 — *A `time_0` variable should be provided if the initial reset or initialization condition is not visible to the properties.*

In many protocols, there are properties that must hold from the moment reset is de-asserted to the occurrence of some condition. In some formal tools, the reset sequence may be handled in a separate initialization procedure, in which case the formal verification engines and the assertions/assumptions may not see the deactivation of the reset signal. If an assumption uses this deactivation to trigger, it may not see the deactivating transition and thus not trigger. Since that assumption is not applied, it can lead to under-constrained inputs and a potential false negative failure of assertions or even a dead end.

A solution is to provide a variable (e.g., called `time_0`) that is initialized to one then reset to zero on the first clock tick, retaining zero thereafter. Assertions that must hold between a reset (time zero) and some other condition can be conditioned either by the de-assertion of reset or by the fact that `time_0 == 1`.

Example 7-3. Detecting Initial Time

```
bit checker_time_0 = 1'b1;
always @(posedge clk)
   if (reset) time_0 <= 1'b1;
   else time_0 <= 1'b0;
```

```
a: assert property(
   @( posedge clk) disable iff (reset)
   ( ((en_vector != 0) ##1 (en_vector == 0)) or
    sva_checker_time_0 )
        |-> (en_vector == 0)[*1:max_quiet] ##1
            ($countones(en_vector) == 1) )
else ...;
```

An alternative is to write a separate assertion that covers only the behavior starting at time zero. This assertion should be placed in an *initial* block so as to evaluate only once starting at the first clock tick.

Rule 7-9 — *All auxiliary state variables shall be initialized.*

An uninitialized auxiliary state variable may cause a mismatch between simulation and formal tools due to differences in interpreting the initial unknown value *'x*. The initialization should be done both in an initial block and also under reset conditions, as illustrated in Example 7-3 on the *time_0* variable.

Rule 7-10 — *Referring to past values at time < 0 shall be avoided.*

A property that is activated by observing the past value of a variable using *$past*, *$rose*, *$fell* and *$stable* over an expression *expr* may not correctly trigger at the first clock tick because the preceding sampled value of *expr* is unknown. The behavior of a simulator and a formal tool may then differ.

A possible solution is to shift the antecedent of the property by one clock tick (or more) forward as shown in Example 7-4 in the case of entering *state == IDLE*.

Example 7-4. Avoiding Initial Unknown Value

```
M3: assert property (
     @(posedge clk) disable iff (!reset_n)
     $rose(state == IDLE)
       |->
##[min_req_latency:max_req_latency] req );

Change to:

M3: assert property (
      @(posedge clk) disable iff (!reset_n)
      !(state == IDLE) ##1 (state == IDLE)
        |->
          ##[min_req_latency:max_req_latency] req );
```

Recommendation 7-11 —*Non-synthesizable code should be enclosed in `'ifndef SYNTHESIS` blocks.*

If the formal tool generates warnings about the presence of `initial` blocks, the number of such messages can be reduced by eliminating the non-synthesizable code by defining the standard macro `SYNTHESIS`.

Recommendation 7-12 —*A property should not be stated solely over an auxiliary state variable value if the property is to be used as an assumption on DUT inputs.*

As illustrated in Example 7-5, this type of assertion cannot be used effectively as an assumption because it requires constraining the DUT inputs driving the auxiliary variable `tmp` in the past clock tick. Depending on the tool and the complexity of the assumptions, this usage may not be possible with random simulation constrained by assumption properties. The result could be that no constraint is imposed at clock tick `t`, leading to a potential inconsistency and dead end at tick `t+1` if `(!tmp == 1)` does not hold. See the footnote on page 291.

Example 7-5. Assumption Over Auxiliary Variable

```
Let port_A and port_B be the input ports of the DUT
that are to be constrained by the assumption.
logic [bw-1:0] tmp = '1; //auxiliary state variable
always @(posedge clk)
   tmp <= c1 ? port_A :
              c2 ? port_B: tmp;
property p;
   @(posedge clk)(| tmp) == 1'b1;
endproperty : p
a: assume property (p);
```

Consider modifying the assertion—and breaking it into two simpler properties—as shown below.

Example 7-6. Assumption Avoiding the Use of Auxiliary Variable

```
property p1;
   @(posedge clk)
   c1 |-> (| port_A) == 1'b1;
endproperty : p1
property p2;
   @(posedge clk)
   !c1 && c2 |-> (| port_B) == 1'b1;
endproperty : p2
a1: assume property (p1);
a2: assume property (p2);
```

The constraints on *port_A* and *port_B* are applied in the current clock cycle rather than in the past one.

Recommendation 7-13 —*Signals that are inputs to the design should not appear as arguments in a sequence that is used with the* `matched` *or* `ended` *property in the consequent of property* `property` *if it is to be used as an assumption on DUT inputs.*

The problem is that this form of assertion may not be used effectively in random simulation under assumption properties because it may require constraining inputs in past clock cycles if no present instance of design inputs is in the assertions. For example, let *a* and *b* be design inputs to be controlled by the assumption *c*, as shown in Example 7-7.

Example 7-7. Use of *ended* in a Consequent of an Assumption

```
sequence s1;
   @(posedge clk) a ##1 b;
endsequence : s1
property p1
   @(posedge clk) c |-> ##1 s1.ended;
endproperty : p1
a1: assume property (p1);
```

Property *p1* should be re-written without using *ended*. From Example 7-7, the solution is simple due to the *##1* delay before *ended*, as shown in Example 7-8.

Example 7-8. Avoiding the Use of *ended* in a Consequent of an Assumption

```
sequence s1;
   @(posedge clk) a ##1 b;
endsequence : s1
property p1
   @(posedge clk) c |-> s1;
endproperty : p1
a1: assume property (p1);
```

In general, the transformation may not be as simple. It may be preferable to approach the problem differently right from the start rather than trying to rewrite the assertion later.

Recommendation 7-14 —*Signals that are inputs to the design should not appear only as arguments of the `$past` system function in a property if it is to be used as an assumption on DUT inputs.*

This form of assertion may not be used effectively in random simulation as assumption properties because it may require constraining inputs in past clock cycles if no present instance of design inputs is in the assertions. For example, let *a* and *b* be design inputs and let *c* be an output. The assertion in Example 7-9 is not causal because the occurrence of the output *c* at the present time is constraining the values of the inputs *a* and *b* in the preceding clock cycle. The cause follows the effect. Although this form can be used as an assumption for formal tools, it cannot be effectively used as a constraint in random simulation because it requires constraining inputs in the past clock cycle. Most likely, this property should be an assertion on *c* rather than an assumption on *a && b*.

Example 7-9. Assumption with *`$past`* over Input Signals

```
property p;
   @(posedge clk)
   c |-> $past(a && b);
endproperty : p
a: assume property (p);
```

When writing assumptions, the sequence expressions should be projecting values on inputs forward in time using the # and * operators rather than using the $*past* operator[3].

Recommendation 7-15 —*Auxiliary state variables should be used.*

To maintain simplicity and reduce the possibility of dead-end situations when modeling assertions for interface protocols, it may be preferable to provide auxiliary finite state machines that track the protocol. The assertions or assumptions can often become simple combinational properties dependent on the state of the machine. This is illustrated next on a simple example.

3. A dead-end avoidance procedure can resolve many issues relative to constraining signal values in past clock cycles. However, the complexity of the assertions may limit the usability of the algorithm. Thus, it is advisable to respect Recommendation 7-12, Recommendation 7-13 and Recommendation 7-14.

For example, a handshake between two entities, as shown in Figure 7-1, can use the auxiliary state machine shown in Figure 7-2 to tracks the protocol. Example 7-10 shows the assertions and assumptions for this protocol.

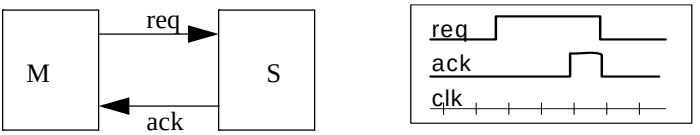


Figure 7-1. Master-Slave Handshake Protocol

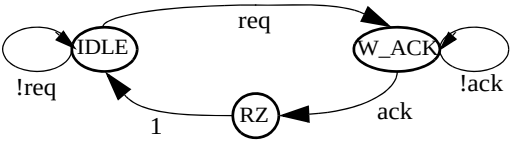


Figure 7-2. Auxiliary State Machine

Example 7-10. Handshake Protocol with an Auxiliary State Machine

```
// MasterCheck
  M1: assert property (@(clk)
        state == W_ACK |-> req );
  M2: assert property (@(clk)
        state == RZ |-> !req );
//SlaveCheck
  S1: assert property (@(clk) // bounded liveness
        (state == IDLE) && req
          |->
       ##[min_ack_latency:max_ack_latency] ack );
  S2: assert property (@(clk)
        (state == I) || (state == RZ) |-> !ack );
```

ASSERTIONS ON DATA

The following guidelines will help to write assertions on data that can be formally verified. Because the use of local variables places certain limitations on the suitability of the assertions for formal verification, guidelines with and without local variables are outlined in separate sections.

Without Local Variables

Without the use of local variables, task calls and non-synthesizable boolean expressions and functions, all SystemVerilog properties are synthesizable into RTL code. However, without such variables, how can the correctness of data values that are carried over time be verified?

There is a simple solution available when there are no overlapping transactions to be verified. Over the span of the assertion evaluation attempt there is no new value introduced. Regular variables can thus be used to store the value to be carried over time. This usage has been illustrated in Example 3-26.

In the case of overlapping evaluation attempts, there are two techniques for avoiding local variables. The first is to use value enumeration and the other is to use specialized static data structures. The examples that follow each guideline in this section are first specified using local variables and then replaced by a form that does not require local variables.

Recommendation 7-16 —*Enumeration should be used for small data value ranges.*

The application of this guideline is specific to the behavior to be specified. It is limited to relatively simple situations involving small data ranges. The original assertion is replicated for all possible values of the variable. In Example 7-12, the assertion has been transformed into 256 assertions, one for each possible value of *data_in*.

Example 7-11. Data transformation verification

```
property data_check_p;
  logic [7:0] v_data_in;
  @(posedge clk) disable iff (reset)
  (all_empty && port_in.accepted_in,
   v_data_in=port_in.data_in)
     |->
       ##4 (port_out.data_out == (v_data_in << 1));
endproperty : data_check_p
data_check: assert property (data_check_p);
```

Example 7-12. Parity In Is the Same as Parity Out, Using Enumeration

```
genvar i;
  generate
    for (i=0; i<=255; i=i+1) begin : forall_7_12
      property p;
        @(posedge clk) disable iff (reset)
        (all_empty && port_in.accepted_in &&
```

```
            (i==port_in.data_in))
             |->
               ##4 (port_out.data_out == (i << 1));
        endproperty : p
        data_check: assert property (p);
      end : forall_7_12
endgenerate
```

In Example 7-11, the variable *v_data_in* could be replaced by using a *$past(port_in.data_in, 4)* because the test of the output data happens a fixed number of clock cycles after the input value is available. If the number of clock ticks is variable or even unknown (e.g. in an open-ended interval), the solution using *$past* may not be suitable due to the large number of registers required to implement the assertion.

Example 7-13 and Example 7-14 verify that a *data* transformation is correct. It is assumed that a particular data tag *id* value is not reused until the same *id* exits the design with the transformed data, validated by *out_ready* (this property should be verified by a separate assertion).

Example 7-13. Checking Tagged Data with Local Variables

```
property p;
  logic [7:0] v_data_in;
  logic [1:0] v_id_in;
  @(posedge clk) disable iff (reset)
    (port_in.accepted_in,
     v_data_in = port_in.data_in,
     v_id_in = port_in.id_in) ##1
       (port_out.accepted_out &&
        (port_out.id_out==v_id_in)
       )[->1]
    |->
      (port_out.data_out ==  (v_data_in + v_data_in) );
endproperty : p
overlap_data_check: assert property(p);
```

Example 7-14. Checking Data with *id* Enumeration and an Array for Data[4]

```
genvar i;
generate
  for (i=0; i<4; i=i+1) begin : forall_id_7_14
```

4. In the assignments to *v_data_in* in the *always* block, the inputs appearing on the right-hand side of the assignments should be first sampled at *#1step* in a *clocking* block. This is to avoid mismatch between the values observed by the assertions that sample signals in the preponed region and by the variables assigned in the *always* block.

```
        logic [7:0] v_data_in;
        always @(posedge clk) begin : each_id
          v_data_in <= reset ?
            0 :
            port_in.accepted_in && (port_in.id_in == i) ?
              port_in.data_in :
              v_data_in;
          data_check: assert property (
            disable iff (reset)
            port_out.accepted_out && (port_out.id_out==i)
              |->
                (port_out.data_out == (v_data_in<<1)) )
          else ...;

       // An auxiliary property to verify that an id value
       // is not reused before exiting on port_out
          id_check: assert property (
            port_in.accepted_in && (port_in.id_in == i)
              |=>
                  !(port_in.accepted_in &&
                  (port_in.id_in == i))
                    throughout
                  (port_out.accepted_out &&
              (port_out.id_out==i))[->1] )
          else ...;
        end : each_id
      end : forall_id_7_14
    endgenerate
```

The original assertion Example 7-13 is replicated into four assertions, one for each possible value of *tag_in*. Each enumerated assertion is much simpler to prove than the original one. Since at most four *data_in* values can be processed by the DUT at any time, the array *data* needs only four elements to store the input data for later comparison. If, by analyzing the design, it can be determined that the treatment of the tag values is symmetrical for all the four values, it may be sufficient to prove only one of the assertions.

Furthermore, if it can be determined by analyzing the structure of the RTL code that the output value function (in this case just a copy of the input value) is symmetrical for all values, it may be sufficient to prove the property for one *data_in* value only. This proof can be achieved by using another *generate* loop over the possible

values of *j* and then proving only one of the generated 256 assertions as shown in Example 7-15, or simply by constraining the input to a constant value.

Example 7-15. Enumeration Over Tag and Data Value Ranges

```
genvar i, j;
logic [7:0] sum_data; // result predictor
always @(port_in.data_in)
  sum_data = (port_in.data_in<<1);
generate
  for (i=0; i<4; i=i+1) begin : forall_id_7_15
    for (j=0; j<256; j=j+1) begin : forall_data_7_15
      overlapped_data_check: assert property(
        @(posedge clk) disable iff (reset)
        port_in.accepted_in &&
        (sum_data == j) &&
        (port_in.id_in == i) ##1
          (port_out.accepted_out &&
           (port_out.id_out==i) )[->1]
            |->
              (port_out.data_out == j) )
       else ...;
    end : forall_data_7_15
  end : forall_id_7_15
endgenerate
```

Example 7-16 shows that enumeration can also be used over the position of individual data bits in addition to data values.

Example 7-16. Enumeration of Bit Index Values

```
genvar i, j, k;
logic [7:0] sum_data; // result predictor
  sum_data = (port_in.data_in<<1);
always @(port_in.data_in)
generate
  for (i=0; i<4; i=i+1) begin : forall_id_7_16
    for (j=0; j<8; j=j+1) begin : forall_index_7_16
      for (k=0; k<2; k=k+1) begin : forall_bit_7_16
        overlapped_data_check: assert property(
          @(posedge clk) disable iff (reset)
          port_in.accepted_in &&
          (sum_data[j] == k) &&
          (port_in.id_in == i) ##1
            (port_out.accepted_out &&
              (port_out.id_out==i))[->1]
            |->
              (port_out.data_out[j] == k) )
         else ...;
      end : forall_bit_7_16
```

```
      end : forall_index_7_16
    end : forall_id_7_16
endgenerate
```

With Local Variables

The introduction of local variables into property specifications may create assertions with dynamic allocation of variables. Without the knowledge of the dynamic behavior of the signals sampled by the assertion, it may be impossible to predict the storage requirements for the assertion during the lifetime of evaluation threads and attempts. However, this knowledge is a requirement for constructing the equivalent synthesizable RTL code.

The problem of determining the amount of storage for local variable instances at compilation time is caused by unbounded non-deterministic branching involving local variable assignments. Unbounded non-deterministic branching can be created by *[M:$]* intervals in *##* delays or repetition, if the operand contains an assignment to a local variable. The simplest solution to controlling the storage needs is to require finite branching. The amount of storage can then be determined at compile time.

In the following example, the objective is to count the number of clock cycles when *c* is asserted between the occurrence of *a* and *d*, as illustrated in the timing diagram in Figure 7-3. Sampling is assumed to be on *posedge clk*.

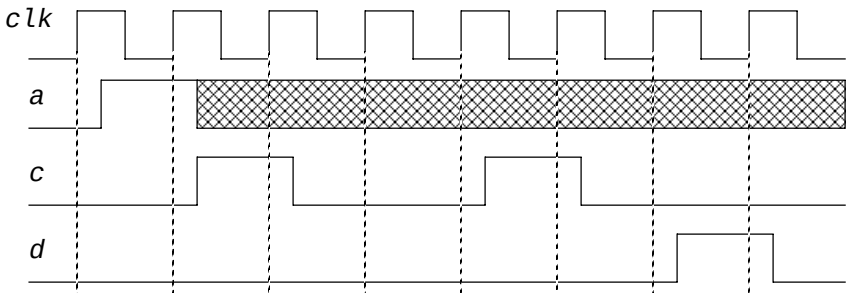


Figure 7-3. Waveform for Example 7-17

The shaded part of the waveform for *a* indicates that after the first assertion of *a*, some further assertions of *a* may or may not occur, as discussed in the examples that follow. In an evaluation attempt triggered by the first assertion of *a*, the value of the

local variable *x* in Example 7-17 would be two when the consequent is evaluated (when *d* is asserted).

Example 7-17. Example of Unbounded Branching with Variable Assignment

```
property p1;
  bit [7:0] x;
  @(posedge clk)
    (a, x = 0) ##1 (!c[*0:$]) ##1
    ( ((c, x = x+1) ##1 (!c[*0:$]))[*0:$] ) ##1 d
      |->
        (x <= 8'd10);
endproperty : p1
```

It is impossible to determine how many threads that increment *x* may become active simultaneously. All of these threads, if they match on *d*, must then satisfy the consequent of the implication for the evaluation attempt to succeed. Furthermore, unless it is known that there can be no new match on *a* during one attempt of evaluation, it is impossible to know how many copies of *x* are needed globally in the multiple overlapping evaluation attempts.

If the user knows that the first *d* that occurs after *a* should terminate the search, the property could be rewritten so that the number of copies of *x* needed in one attempt is bounded:

Example 7-18. One Copy of *x*

```
property p2;
  bit [7:0] x;
  @(posedge clk)
    (a, x = 0) ##1
    (
      !d throughout
          (!c[*0:$]) ##1
          ( ((c, x = x+1) ##1 (!c[*0:$]))[*0:$] )
    )
    ##1 d
      |-> (x <= 8'd10);
endproperty : p2
```

There are at most two copies of *x* needed in each evaluation attempt. One to continue counting and one holding the last value while *d* is tested whether to compare *x* with *expr* or to just stop that thread. However, careful analysis is still required to ascertain that the expression *!d* is actually the complement of *d*. A more direct

solution is to use the *if-else* statement together with recursion, as shown in Example 7-19.

Example 7-19. Using *if-else* and Recursion

```
property p3;
  int x;
  @(clk) (a, x = 0) |=> p_rec(x);
endproperty : p3
property p_rec(x);
  @(clk)
  if (d)
    (x <= 8'd10)
  else
    if (c) (1, x=x+1) |=> p_rec(x)
    else   1 |=> p_rec(x) ;
endproperty : p_rec
a7_19: assert property (p3);
```

There remains the question about how many instances of *x* are needed between multiple overlapping attempts. If there is no knowledge that *a* cannot occur again before *d*, it is impossible to know how many instances of *x* are needed.

The possible approaches to the problem may be as follows:

1. The user indicates an upper bound on the number of concurrent attempts that require local variable allocation. If the bound is violated, the property implementation in RTL would fail.
2. Design code is analyzed to determine that *a* cannot occur again before *d*.
3. Reformulate the property to indicate explicitly that a single attempt is possible at any time.

Approach one depends on whether the users can accept such a requirement and does not require the tool to analyze the property.

Approach two depends on the ability of the tools to analyze the DUT at the same time as constructing the RTL code for the property. This ability of the tool cannot be predicted, and also it couples the property tightly to the particular DUT, thus limiting reuse.

Approach three is probably the most useful one.

In Example 7-20, the evaluation attempt would terminate vacuously when *a* becomes true a second time before *d*. In Example 7-21, the additional unwanted (!) occurrence of *a* is checked by a separate property.

Example 7-20. Using Repetition, Vacuous Success on an Additional *a*

```
property p4;
  int x;
  @(clk)
    (a, x = 0) ##1
    (
      !d && !a throughout
         (!c[*0:$]) ##1
         ( ((c, x = x+1) ##1 (!c[*0:$]))[*0:$] )
    )
    ##1 d
       |-> (x <= 8'd10);
endproperty : p4
a7_20: assert property (p4);
```

The boolean expressions *a*, *b*, *c* and *d* must be analyzed to determine that no new evaluation of the property is possible while an evaluation attempt is in progress.

Example 7-21. Signaling Failure if Another *a* Before and Including *d*

```
a7_21: assert property (
  @(posedge clk)
  (a |=> !a throughout (d[->1]) )
);
```

Once the assumption on *a* and *d* are in place and verified by other assertions, it is easier to use a static counter and simple assertions as shown in Example 7-22.

Example 7-22. Using a Static Counter to Count *c*'s

```
localparam MAX_C = 8'd11; // value > exp
logic [7:0] x = MAX_C;
always @(posedge clk)
  x <= d || a?
     0 :
     c && (x<MAX_C) ?
       x+1 :
       x ;
property p5;
  @(posedge clk)
    d || c |-> (x < 8'd10);
endproperty : p5

/* The assertion will fail on any c that exceeds the
```

```
maximum of 8'd10. This is unlike the other assertions
where a failure is detected only when d occurs and the
count of c's exceeds the maximum. It is left to the
reader to modify this assertion or the others so that
all have the same reporting - either all when d occurs
or whenever c occurs that exceeds the maximum count.*/

a7_22: assert property (p5);
```

This solution also lets the property be a constraint on `c` (or `d`), which the preceding solution using local variables may not do without additional processing of potential dead ends.

In finite branching (outside an indefinite loop) such as in `[*M:N]`, the number of local variable instances can be determined by analyzing the branching structure within the repetition. With the current knowledge of the property structure and potential solutions, the recommendations for writing “synthesizable” properties with local variables are as follows.

Let `p` be a property and `p_head` and `p_tail` be subsequences of the property such that if `p_head` matches, then `p_tail` is evaluated.

Rule 7-17 — *A variable assignment shall not occur on the right-hand side of ##[M:$] delay.*

If there is a variable assignment reachable from the right-hand side of a `##[M:$]` delay, then this unbounded non-determinism will generate an unknown number of copies of the variable. A possible solution is to exit with one choice only, e.g., the delay must be specified by using `[->1]` go-to repetition with a Boolean operand instead of a ## delay.

Rule 7-18 — *[*M:$] repetition shall have a deterministic exit.*

If the loop involves a variable assignment or if there is a variable assignment following the loop, again the number of copies of the local variable may be impossible to determine. Therefore, there must be only a single exit from the loop, i.e., the choice to loop or not must be deterministic.

Rule 7-19 — *s1 within s2: There shall be no variable assignment in s1.*

The *within* operator is derived using basic operators that involve *##[0:$]* preceding *s1*. This derivation may make it impossible to determine the number of required copies of the local variable, as specified in Rule 7-17.

Rule 7-20 — *If `p_tail` contains a local variable assignment, then `p_head` shall involve only finite branching constructs.*

Example 7-23 illustrates such a structure.

Example 7-23. Finite Branching Before Local Variable Assignment

```
property p;
   int id; int data;
   @(clk)
      s1 ##[M1:N1] (ready_in, id=id_in, data=data_in)
        |->
          ##[M2:N2] ready_out &&
                    (id == id_out) &&
                    (data_out == fnct(data));
endproperty : p
```

Whenever `s1` matches and is followed by `ready_in` within `[M1:N1]` clock ticks, the `id_in` and `data_in` signals are sampled and then, within `[M2:N2]` clock ticks, `ready_out` must be asserted and `id_out` must match the value of stored `id_in`. The output data `data_out` value must be equal to a function *`fnct`* of the stored `data_in`. The number of instances of the variables `id` and `data` is at most `N1-M1+1` in a single match of `s1`. Note that once `s1` matches, then the consequent must match on all occurrences of `ready_in` within the `[M1:N1]` interval.

The RTL implementation of the property in Example 7-23 may be obtained using a shift register of length `M1-N1+N2` stages.

Compatibility with Formal Tools

In designing assertion-based verification IP, care must be taken to use synthesizable assertion statements and formulate them so that the appropriate set of assertions can be used as assumptions, as described earlier in this chapter and in Chapter 3

The VMM Checker Library has been constructed so as to operate with formal tools with minimal additional requirements. No local variables are used. Also, for any range coverage using a *`covergroup`* at Level two, there are Level three *`cover property`* statements on the limit values of the range. These statements can be used as search goals by a formal tool.

SUMMARY

In this chapter, the rules and recommendations for using assertions with formal tools have been outlined. Since formal tools are generally more restrictive in the kind of models they can effectively use, it is likely that some of the rules may be relaxed or tightened, depending on the specific tool. Therefore, the contents of the chapter should be interpreted as providing better understanding of the issues involved and make the user aware that not all assertion forms may be usable with all formal tools. Similar restrictions will likely apply to assertions used in a synthesizable form in emulation and hardware prototyping.

Summary of basic rules:

- Keep assertions simple.
- Use auxiliary state variables.
- Initialize all variables.
- Handle initial reset that may not be seen by assumptions.
- Avoid constraints on past values.
- Use no unbounded (or large bounded) intervals in antecedent or consequent of implication.
- Use deterministic choices whenever possible (go-to repetition or if-else rather than ##).
- Limit the use of local variables (if supported at all).
- Avoid end-to-end properties over large blocks.
- Separate properties by signal direction to use as assertions or assumptions.

CHAPTER 8 SYSTEM-LEVEL VERIFICATION

A system is a design that is composed of independently designed and independently verified blocks as well as a block interconnect infrastructure. System-level verification is the verification of the correct interaction among these individual blocks. Each block, including the block interconnect infrastructure—a bus and associated bridges and other supporting elements—has been verified individually. Therefore, system-level verification should focus on the functionality embodied by the combination of the blocks. Any functionality that is entirely contained within a block is better verified at the block level.

System-level verification can take place on systems of any size, from tens of thousands to tens of millions of ASIC gates. In all cases, the principal requirement of verifying that the specified functionality and performance goals have been met is the same. The system architect establishes performance, latency and bandwidth goals for the various components of the system. The implementation teams must meet these goals. The verification teams, along with demonstrating that these goals are met, must verify all corner cases introduced by the implementation to demonstrate that an invalid state or behavior cannot be reached.

The biggest challenge facing a system design team is not in the specification and integration of many design blocks. It is in achieving a confidence level in the correctness of the final design. This chapter, and its companion Appendix C, describes a methodology and verification tasks applied to system-level verification. This chapter will be of interest to block- and system-level architects and verification engineers, with a focus on system-level integration verification.

Beginning with a discussion of extensible verification components and their co-ordination via a manager component, this chapter then moves onto discussing system-level verification and verifying transaction-level environments. Wrapping up this chapter is a discussion of hardware-assisted verification.

EXTENSIBLE VERIFICATION COMPONENTS

The techniques described in previous chapters suggest the creation of individual transactors that can be reused from block- to system-level environments. When considering these transactors from a system-level perspective, it may be necessary to extend or combine their block-level functionality into system-level functionality. This section describes the how individual transactors can be structured into system-level transactors referred to as an extensible verification component (XVC).

XVCs provide a foundation for modular, scalable and reusable system-level verification environments, with the aim of minimizing test set-up overhead. XVCs can be used to drive block interconnect infrastructures or external interfaces. They can also support other XVC components by monitoring system state and providing notification information.

Methods of generating test vectors to hit code and functional coverage on a DUT have been established now for many years and can be implemented in numerous ways. The purpose of an XVC is to support system-level integration and functional verification using both directed and random testing using a unified methodology approach. The structure of an XVC is such that it is highly portable to different system-level designs and across design abstraction levels.

An XVC can encapsulate verification expertise and system-level functions in a standard way. Despite potentially having widely varying levels of functionality, the user of an XVC is always presented with the same look and feel at a testbench level, which contributes to a reduced learning curve and enabling a consistent system-level test-bench control mechanism.

XVC Architecture

An XVC is a container for verification IP divided into two elements. The top layer is a user-extensible library of test actions with a defined action interface for integration into a verification environment. The bottom layer integrates individual transactors for implementing the actions on a physical- or transaction-level interface.

The action interface of the top layer allows coordination of test stimulus. The top layer verification environment interface of an XVC can connect directly to an XVC manager. An XVC manager can support any number of XVCs at one time and uses this interface to schedule execution of individual actions within any given XVC. XVCs can also pass data and status back up to the test control infrastructure to communicate with other XVCs in the same environment.

As illustrated in Figure 8-1, an XVC has two layers. The generator layer executes user-defined actions. The driver layer interacts with the DUT to execute or monitor the transactions as required by the action. The generation layer controls the transactors in the driver layer. The action library contains a selection of known test actions that can be executed by the XVC.

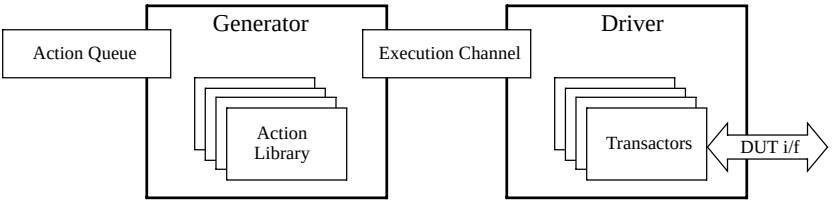


Figure 8-1. Structure of an XVC

Recommendation 8-1 — *XVCs should be under the control of a single XVC manager.*

Experience shows that most of the system verification issues are observed when blocks compete for common system resources. To create the necessary competitive scenarios, it is not sufficient to simply execute a single-threaded test. Similarly, it is unlikely that independent stimulus streams on independent external interface will create the required scenarios. Verification components must therefore be coordinated to create the required stimulus. This coordination is easiest to achieve when a single, central controller coordinates the execution of multiple XVCs.

See “XVC Manager” on page 316 for more details on XVC managers.

Rule 8-2 — *XVCs shall be configurable to match the design they are testing.*

Even though designs may be implemented using the same interface protocols, there may be differences in how the protocol is physically implemented by different designs. Parameters such as bus width, FIFO depth, clock frequency, optional side band signals shall be configurable in the XVC.

Rule 8-3 — *XVCs shall be configurable to constrain functionality as required.*

When a DUT implements only a subset of a particular interface protocol, a corresponding driver XVC shall be configurable to prevent certain stimulus from being generated.

Recommendation 8-4 — *XVCs should be configurable to allow for error injection.*

As well as testing for known good conditions within a DUTs state space, the verification requirements may require that invalid protocol or data be applied to test the error detection and recovery mechanisms of the DUT.

Rule 8-5 — *Stimulus XVCs shall support directed stimulus and constrained-random stimulus generation.*

Using directed stimulus may cover a large area of state space. However, interesting corner case states can be automatically explored by the ability to apply constrained-random stimulus as part of the action library.

Conversely, an environment based on constrained-random stimulus may not be able to reach specific corner cases. It may be necessary to resort to directed stimulus to meet the more stringent stimulus requirements.

Recommendation 8-6 — *Protocol-checking XVCs should be separate from driving and or monitoring XVCs.*

When error injection is disabled, the correctness of the protocol must be maintained. A pure protocol-checking XVC should be developed for this purpose. A pure protocol-checking XVC can be used in different system verification environment configuration independently of the nature of the protocol agents.

Recommendation 8-7 — *Coverage collection XVCs should be separate from driving and or monitoring XVCs.*

Although it is possible for an XVC implementation to combine a driver, a protocol checker, a scoreboard and coverage monitor, practical experience has proven that a more effective approach would be to implement single-purpose XVCs to cover these functions independently. These individual XVCs can then be more easily combined, reused and maintained. Information can be shared among XVC functions via a common data structure.

For example, notification descriptors can be exchanged between XVCs with coverage or scoreboard related information contained within them. These descriptors can be used to develop reactive test scenarios where different stimulus behavior patterns are triggered depending on observed behavior or coverage feedback.

Suggestion 8-8 — *An XVC may not have a need to connect to a DUT.*

There is no prescribed way of implementing functionality within a specific XVC. An XVC need not always connect to a device under test. A passive XVC may just have a generator layer that acts as a counter or coverage collection or scoreboarding component.

Implementing XVCs

XVCs are implemented by defining their driver layer using individual transactors. Which transactors are used is application-specific and beyond the scope of this section. Actions that make use of the functionality provided by the driver layer can then be predefined for that XVC. Additional actions can be further defined by users.

Appendix C specifies a set of base classes available to help in implementing interoperable XVCs. Figure 8-2 shows a conceptual relationship between the classes associated with implementing an XVC.

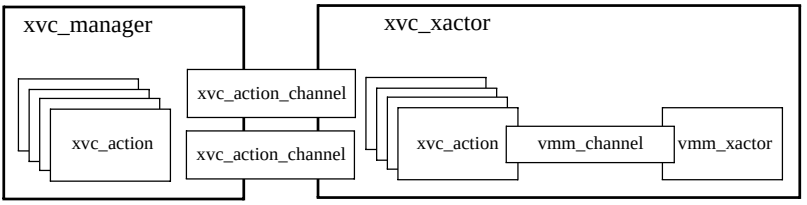


Figure 8-2. VMM Classes Used for Building an XVC

The XVC generator layer is the main engine of the XVC. It is able to parse action commands into action descriptors and interpret these descriptors to execute the described actions. For example, an Ethernet source XVC may have separate actions to configure the generator, start the generation and stop the generation. The XVC driver layer is the part of the XVC that implements the actions. It may be implemented using one or more transactors. It may also have a number of different implementation configurations, each communicating to different abstraction levels of the DUT or acting as different types of agents on an interface.

A fully implemented XVC presents a simple interface to the design/verification engineer, leaving the verification engineer to concentrate more on exercising the

functionality of the device under test. This structure reduces the learning curve for verification engineers who wish to reuse the XVC.

The remainder of this section provides guidelines for implementing an XVC using the the base classes provided in VMM library and described in Appendix C.

Rule 8-9 — *XVCs shall be derived from the* `xvc_xactor` *base class.*

The `xvc_xactor` base class, in collaboration with the `xvc_action` base class, provides a common baseline functionality. To inherit and take advantage of the XVC base functionality and to promote a common coding standard and reuse, each XVC must be derived from the same base class.

Example 8-1. Implementing an XVC

```
class ahb_master_xvc extends xvc_xactor;
   ...
endclass: ahb_master_xvc
```

Rule 8-10 — *XVCs shall contain an execution channel instance in the* `xvc_xactor::exec_chan` *class property.*

To connect the XVC generator layer to the XVC driver layer, an internal execution channel needs to be created. This channel will be used by action descriptors to execute the action.

Example 8-2. Instantiating an Execution Channel

```
class ahb_master_xvc extends xvc_xactor;
   ...
   function new(...);
      ...
      super.exec_chan = new(...);
      ...
   endfunction: new
   ...
endclass: ahb_master_xvc
```

Rule 8-11 — *Transactor instances used to implement the XVC driver layer shall be stored in the* `xvc_xactor::xactors[]` *class property.*

An XVC driver layer is required to translate transaction information into stimulus or from response that is compatible with the DUT. This translation is performed by one or more transactors with the top-most transactor connected to the execution channel in the `xvc_xactor::exec_chan` class property.

The transactor instances must be stored in the *xvc_xactor::xactors[]* array to allow the *xvc_xactor* base class to register and unregister any callbacks extensions required by the action execution.

Example 8-3. Storing Transactor Instance References

```
class ahb_master_xvc extends xvc_xactor;
   ahb_master_xactor ahb_master;

   function new(...);
      ...
      this.ahb_master = new(...);
      super.exec_chan = this.ahb_master.in_chan;
      super.xactors.push_back(this.ahb_master);
      ...
   endfunction: new
   ...
endclass: ahb_master_xvc
```

Rule 8-12 — *The xvc_xactor::start_xactor(), xvc_xactor::stop_xactor(), xvc_xactor::reset_xactor() methods shall be extended to call their corresponding methods in the execution transactors.*

Extending these methods will allow the correct starting, stopping and resetting of the XVC. The base class does not automatically invoke these methods in the transactors found in the *xvc_xactor::xactors[]* class property.

Implementing Actions

An XVC action is a high-level operation implemented using the transactors in the driver layer of an XVC. An XVC action is described by an action descriptor. An action descriptor can be manually created and specified, or randomized or set by parsing an action command. Actions always execute in order. Actions may execute atomically or can be interrupted by other actions. Actions may execute in zero-time or may be blocking. Actions may be immediate, completing only once their execution has completed. Others may fork off threads that keep executing, even though the action is nominally completed. Other actions may terminate threads started by previous actions. For example, a start-generation action would execute in zero-time without being interrupted and start a generation thread. A stop-generation action may have to wait for an opportune moment, and then terminate the generation thread started by the start-generation action.

The remainder of this section specifies guidelines for implementing actions using the VMM library as defined in Appendix C.

Rule 8-13 — *Action descriptors shall be derived from the* `xvc_action` *base class.*

The `xvc_action` base class provides common functionality shared by all action descriptors. It is also the type used by the `xvc_xactor` base class to implement generic XVC functionality. To be usable with `xvc_xactor`-based XVCs, action descriptors must be derived from the `xvc_action` base class.

Example 8-4. Defining an Action

```
class ahb_master_config extends xvc_action;
   ...
endclass: ahb_master_config
```

Rule 8-14 — *The* `xvc_action::parse()` *method shall be extended.*

This method lets actions be defined using commands. The syntax of the command used to specify a user-defined action is defined by this method. This method is invoked by the `xvc_xactor::parse()` method when parsing a command based on known actions.

Rule 8-15 — *If the parsed command is invalid, it shall be silently ignored.*

If the syntax is not recognized, the command shall be ignored without producing an error message. The command may be a valid command for another action. If an error message is produced, the simulation will be considered failed even though the command was valid for another action.

If a command is not recognized by any of the known actions, the `xvc_xactor::parse()` method will issue an appropriate message.

Rule 8-16 — *The* `xvc_action::parse()` *method shall return an action descriptor instance corresponding to the parsed command or* `null`.

If the command is successfully parsed, the corresponding action descriptor is returned as a new instance of the action descriptor class. Any parameter is set according to the values specified in the command.

If the command is not recognized, `null` is returned.

Example 8-5. Parsing an Action Command

```
class ahb_from_file extends xvc_action;
   string fname;
   ...
   virtual function xvc_action parse(string argv[]);
```

```
            parse = null;
            if (argv.size() != 2) return;
            if (argv[0] != "read") return;
            begin
               ahb_from_file act = new;
               act.fname = argv[1];
               parse = act;
            end
         endfunction: parse
         ...
      endclass: ahb_from_file
```

Rule 8-17 — *The `xvc_action::execute()` method shall be extended.*

The actual execution of the action is performed when the XVC invokes the action descriptor's *`execute()`* method. The user-defined implementation of this method defines the execution of the action.

This method interprets any argument or parameters in the action descriptor instance and executes the action accordingly.

Example 8-6. Definition of an Action Execution

```
class ahb_from_file extends xvc_action;
   string fname;
   ...
   virtual task execute(...);
      int fp = $fopen(fname);
      if (fp == 0) ...
      ...
   endtask: execute
   ...
endclass: ahb_from_file
```

Rule 8-18 — *Actions shall run via the execution channel.*

The execution channel, found in the *`xvc_xactor::exec_chan`* class property, is provided to the action execution via the *`exec_chan`* argument of the *`xvc_action::execute()`* method. While executing the action, transactions can be supplied to or received from the XVC driver layer via that channel.

Example 8-7. Executing an Action

```
class ahb_from_file extends xvc_action;
   string fname;
   ...
   virtual task execute(vmm_channel exec_chan,
                        xvc_xactor  xvc);
      ...
      ahb_tr = new;
      ahb.kind = WRITE;
      ...
      exec_chan.put(tr);
      ...
   endtask: execute
   ...
endclass: ahb_from_file
```

Example 8-7 shows an atomic immediate action. A more autonomous XVC may require using state-based or command actions. State-based or command actions modify the state of an autonomous XVC. They may start, modify or terminate independently executing threads. For example, a generator XVC could implement configure, start-generation and stop-generation actions.

Alternative 8-19 —*Actions may include callback extensions in their execution.*

The execution channel and the abstraction layer provided by the XVC driver layer may prove insufficient to execute the desired action. Some actions will find it necessary to use callback extensions for one or all of the execution transactors.

Action-specific callback extensions can be automatically registered by the XVC with the appropriate execution transactor if their instance is stored in the *`xvc_action::callbacks`* class property. If an element of the array is not *`null`* (i.e., contains a callback extension instance), then it is prepended to the corresponding execution transactor in the *`xvc_xactor::xactors[]`* class property (e.g., extension found in *`xvc_action::callbacks[0]`* will be registered with *`xvc_xactor::xactors[0]`*). The callback registration occurs before the action's *`execute()`* method is invoked and unregistered when it returns.

Example 8-8. Including Callback Extensions in Action Execution

```
class ahb_with_retries_cbs extends ahb_master_cbs;
   ...
endclass: ahb_with_retries_cbs

class ahb_with_retries extends xvc_action;
   ...
   function new(...);
      ...
      ahb_with_retries_cbs cbs = new(...);
      super.callbacks[0] = cbs;
   endfunction: new
   ...
endclass: ahb_with_retries
```

Alternative 8-20 —*An XVC may support out-of-order action execution via interrupt actions.*

Each XVC may have any number of *interrupt actions*. Interrupts are executed at the earliest opportunity. Actions are scheduled for execution by the XVC manager and can be scheduled as either interrupt actions or normal actions. Actions to be scheduled as interrupt actions are submitted to an XVC via the interrupt channel.

Rule 8-21 — *Non-atomic normal actions shall invoke* the *`xvc.wait_if_interrupted()` method.*

Non-atomic actions are actions that may be interrupted by higher-priority interrupt actions. The granularity of the interruption is defined by the points where the *`xvc.wait_if_interrupted()`* method is invoked. This method must be invoked at all points, in the execution of an action, where it can be safely interrupted.

Example 8-9. Non-Atomic Normal Action

```
class ahb_from_file extends xvc_action;
   ...
   virtual task execute(vmm_channel exec_chan,
                        xvc_xactor  xvc);
      ...
      while (...) begin
         ahb_tr = new;
         ahb.kind = WRITE;
         ...
         xvc.wait_if_interrupted();
         exec_chan.put(tr);
      end
      ...
   endtask: execute
```

```
   ...
endclass: ahb_from_file
```

Rule 8-22 — *Interrupt actions shall not invoke* `the xvc.wait_if_interrupted()` *method.*

Interrupt actions are atomic actions that cannot be interrupted by higher-priority interrupt actions. Any call to the `xvc.wait_if_interrupted()` method is ignored and a warning message is issued.

XVC MANAGER

With the ever increasing design complexity of system-level designs, support for coordination of multiple verification components is essential to provide a working platform from where the construction of complex verification scenarios can begin. The XVC manager is an optional verification component responsible for the high-level synchronization of XVCs. The synchronization and XVC control mechanisms can be user-defined according to the need of the system or a specific test.

Rule 8-23 — *There shall be only one instance of the XVC manager.*

The purpose of the XVC manager is to coordinate XVCs to create relevant conditions for particular tests. Using multiple XVC managers would require an additional level of coordination among the managers.

Recommendation 8-24 —*An XVC manager should be extended from the* `xvc_manager` *base class.*

The `xvc_manager` base class provides generic functionality that is useful for controlling and synchronizing a varying number of XVCs in a system-level verification environment. It is designed to work in collaboration with the features present in the `xvc_xactor` and `xvc_action` base classes.

Example 8-10. Defining an XVC Manager

```
class env_manager extends xvc_manager;
   ...
endclass: env_manager
```

Predefined XVC Manager

A predefined XVC manager that uses test scenario descriptions written as external configuration files in a plain text format is available. These files can be reused or modified to new requirements with relative ease. The ability to direct the test via simple text input files enables the user to quickly achieve effective test results without a detailed understanding of the internals of the verification environment. The predefined XVC manager is specified in “vmm_xvc_manager” on page 444.

Having test scenarios specified as external files also eliminates the need for any re-compilation of testbench components or DUT to run different test scenarios. This specification in turn helps to minimize test turnaround times for large and complex system designs. See “vmm_xvc_manager” on page 444 for a description of the test scenario file syntax.

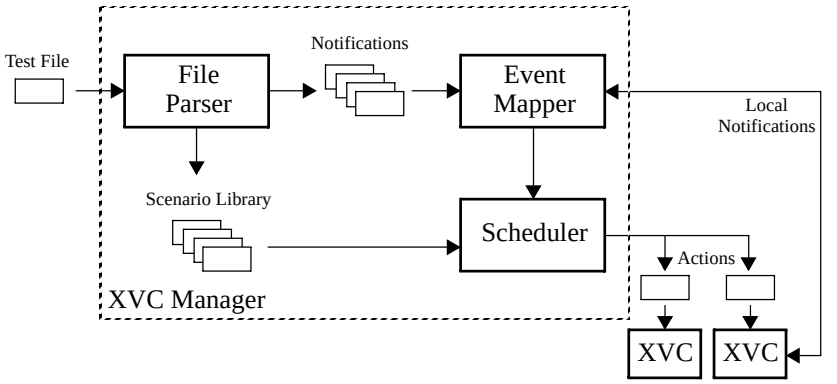


Figure 8-3. Structure of the Predefined XVC Manager

The concept of a *test scenario* is fundamental to the function of the predefined XVC manager. The predefined XVC manager directs XVCs within the scope of a test scenario. A system-level test program can contain one or more test scenarios. Test scenarios are designed to meet one or more verification requirements. A test program might consist of a number of test scenarios which, when taken together, fulfills several test requirements for a DUT. Test scenarios can also be used to encapsulate common sequences of actions so they can be reused by different test programs. For example, the operations required of a number of XVCs to configure a system could be contained in a scenario.

Test scenario files for the predefined XVC manager are fixed prior to run time and cannot be dynamically modified. A test program includes the descriptions of test scenarios and information regarding the order in which each scenario is executed. Scenarios may be repeated any number of times and in any order. Every test action

that can be executed in the test program is checked by its target XVC before executing the first action. This checking prevents a semantic or syntax error in the scenario file from aborting a simulation after a long run.

The remainder of this section specifies guidelines for using the predefined XVC manager as defined in Appendix C.

Rule 8-25 — *The predefined XVC manager shall not be extended in any way.*

The predefined XVC manager is designed to be fully portable and generic from a verification environment to another. There is no need to extend it in any way to specialize it to a particular environment. This continuity is also true for the syntax of the test description file or scenario input file.

Rule 8-26 — *A test scenario shall include one or more XVCs.*

As the predefined XVC manager does not generate stimulus or collect coverage or scoreboard data by itself, there shall be at least one XVC within the testbench environment that is used by the test scenario.

Rule 8-27 — *A test scenario shall end only when all the XVCs associated with it have completed their actions or when a predetermined system notification is indicated.*

When executing a scenario, the predefined XVC manager queues all of the actions scheduled to run on an XVC. Once all XVCs have completed their actions, the scenario ends.

The predefined XVC manager is also responsible for detecting an end-of-simulation condition and stopping the simulation run. The trigger for this condition may be either:

- A state transitioned to a particular system state—the simplest being that the system transitioned from *active* to *idle*
- A specific runtime limit was reached

These conditions are specified in the manager scenario file `STOPON` command. See “Commands” on page 447 for more details on the predefined XVC manager command set.

Rule 8-28 — *A test scenario shall guarantee that a scenario finishes in a timely and orderly manner.*

The user should define scenarios to end at the appropriate time. This definition can be categorized in a number of ways depending on the requirements of the test being implemented:

- When functional coverage goals are met
- When all input stimulus is exhausted
- If some unexpected state or error condition occurred

The predefined XVC manager allows graceful-stop and immediate-stop requests for this purpose.

SYSTEM-LEVEL VERIFICATION ENVIRONMENTS

To fully verify the requirements of a block or a system, common practice is to construct a custom testbench targeted solely at meeting these demands. However, unless a standard approach is taken, block-level testbench components or functional coverage elements cannot easily be reused at the system level.

Rule 8-29 — *A transactor shall be used in place of the CPU or DSP processor.*

This chapter demonstrates using verification components in place of a CPU/DSP to control different system-level verification environments. Using a CPU/DSP in software-driven, system-level verification environments is still very much encouraged, but that usage is covered in detail in Chapter 9.

A transactor presents a direct cycle-level accurate mechanism for driving stimulus into the system without the overhead of programming a processor to perform this task. This transactor provides an easy-to-control bus master. This approach also avoids the verification of both master and slave bus agents in one environment, which can lead to challenging error detection conditions. For example if there were an error in the connection of a master or slave bus agent, then either could be masking the connection issue. A transactor will also be easier to synchronize with other events in the system design or verification environment.

Finally, a transactor can be written to generate a much wider range of protocol values and behaviors over a smaller number of bus cycles, and can therefore be used to achieve better functional coverage.

Recommendation 8-30 —*System-level verification should be split across multiple verification environments.*

For a given system design, its associated functional verification plan specifies system-level verification requirements. It is easier to use a number of different verification environments to meet these requirements. Each environment is concerned with a particular set of orthogonal functional verification goals and is designed to cover those requirements efficiently. The orthogonality of the system-level requirements makes using a single environment needlessly complex and can in fact mask certain system configuration errors.

Figure 8-4 shows different system-level verification environments for the same system. Dotted boxes represent "sockets" where the CPU/DSP and design blocks (peripherals) interface to the block interconnect infrastructure. In all cases, the CPU/DSP has been replaced with a verification component that drives transactions into the system, as per Rule 8-29. In the block interconnect infrastructure environment, even the peripherals themselves have been substituted for verification components. In the other environments, the peripherals are present and verification components are used to drive and monitor stimulus on their external interfaces.

Verification can begin with any of the verification environments shown in Figure 8-4 and at any abstraction level. However, a system design implemented at a given abstraction level may be more or less suited to one verification environment over another. Considering each environment in Figure 8-4 in bottom-up order:

Block interconnect infrastructure environment — As described earlier, the block interconnect infrastructure is first pre-verified in its own verification environment. Verification components substitute for the master and slave peripherals at the points where the peripherals would interface to the bus. The purpose of this verification environment is to check the functional correctness of data transfers, protocol rules and bus performance requirements, such as latency and bandwidth.

Basic integration environment — Integration verification checks the correctness of the connectivity by toggling all I/O ports in a system. Replacement for the various system components with verification components to drive bus transactions is the preferred method of applying integration stimulus. Integration tests are written to ensure that each block has been correctly integrated into the system-level hierarchy. Internal functional behavior of the system components is not verified.

Low-level system functional environment — The purpose of this environment is to cover any functionality that cannot readily be observed from the transactor substituting for the CPU/DSP. Such functionality may include monitoring of control

signals, reset mode checking, or any other functionality that is not easily looped back to the CPU/DSP for read-back or control. As with the basic integration environment, internal functional behavior of the system components is not verified.

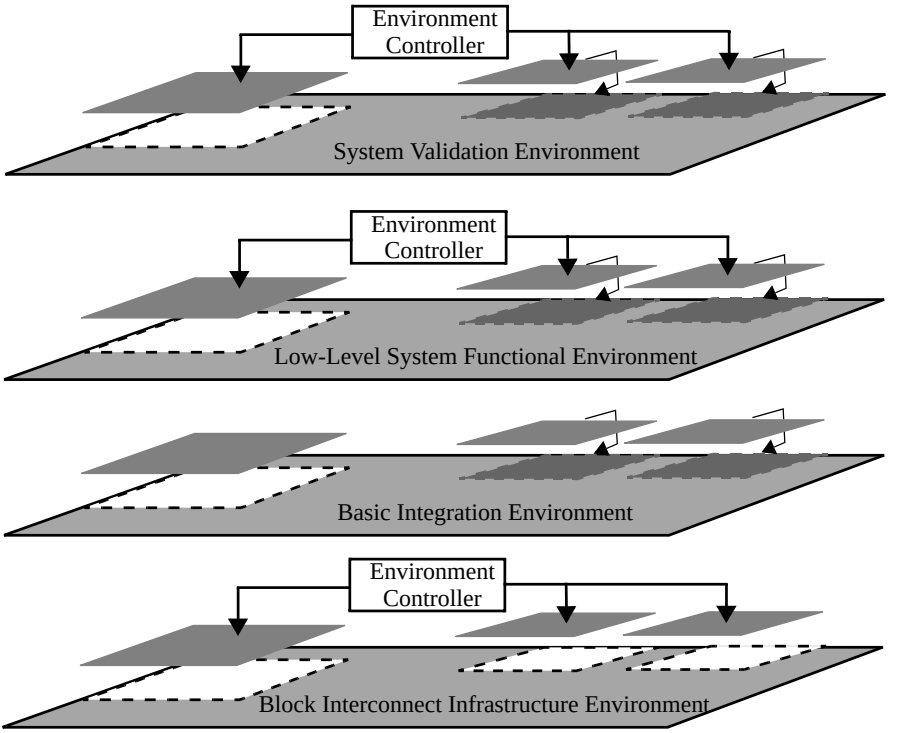


Figure 8-4. System-Level Verification Environments

System validation environment — System validation ensures that the overall performance requirements—such as latency and bandwidth—of the system are met. In this environment, verification components drive or monitor the external interfaces of all blocks in the system. Only the CPU/DSP is replaced with a transactor. This environment should use an abstraction model of the system that efficiently simulates with high throughput with the required level of accuracy to help minimize test turnaround time.

Recommendation 8-31 —*System-level verification environments should reuse block-level verification components where applicable.*

The guidelines outlined in the previous chapters make it possible to architect block- and system-level verification environments so that the effort spent in creating the

different block-level verification components is not discarded when moving to the system level.

Verification components constructed as described in previous chapters can readily be reused at the system level when required. However, instead of being directly controlled, block-level components can be reconfigured to let them perform operations that are more relevant to the system. The reconfigurability of verification components addresses the difference in purpose of the block- and system-level verification.

For example, in block-level verification, tests focus on meeting code and functional coverage requirements of the block under test. These tests create stimulus to exercise corner case states—stimulus that may not be representative of system-level operations or corner cases. When the block is integrated into the system, the verification component may have to be reconfigured to create different stimulus patterns designed to exercises system-level corner cases. For reactive verification components that drive and monitor an interface, it usually involves reconfiguring the component from reactive to passive mode so that only the monitoring function is enabled.

It must be noted that the benefits of reusing such components should not automatically be assumed. They should be considered according to the system-level test requirements. For example, if requirements can be met by simply looping back the external interface of a block, then there is no benefit to reusing the block-level verification component for that interface.

Rule 8-32 — *A watchdog timer shall terminate the simulation in the absence of progress.*

System-level environments must contain a watchdog timer mechanism that will bring the simulation to a halt in a controlled but timely fashion in the event of a lockup in the DUT. The watchdog timer is reset whenever progress is observed by the verification components monitoring simulation progress.

Because of the inherent complexity of the system being verified, system-level simulations are usually lengthy. Simulation resources must be used efficiently to perform as many system-level tests as possible. A watchdog timer will prevent simulations from running without accomplishing any useful work until manually interrupted. Such simulations also waste regression runs by preventing the execution of subsequent tests.

Recommendation 8-33 —*System-level verification environments should be XVC-based.*

Extensible verification components or XVCs are better than raw transactors and generators for system-level verification because XVCs are specifically designed for creating relevant system-level stimulus scenarios. Unlike individual transactions in a low-level transactor, XVC actions are abstractions of lower-level transactors and/or generator behavior. Also, XVCs have a common communications interface and structure that lets their actions be synchronized. See section titled "Verifying Transaction-Level Models" on page 332 for more details on the structure and implementation of an XVC.

Recommendation 8-34 —*AN XVC-based environment should use an XVC manager.*

To be able to drive multiple simultaneous input streams into a system design to emulate real-world operation with minimal set-up cost, an XVC manager should be used to coordinate a group of one or more XVCs via their respective communications interfaces.

Figure 8-5 shows an example of XVC-based verification environment. Each XVC is attached to its corresponding physical interface and is controlled by an XVC manager. See “XVC Manager” on page 316 for more details on the XVC manager.

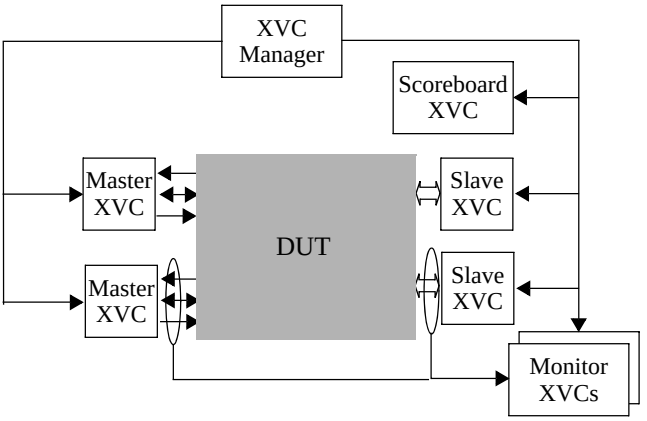


Figure 8-5. XVC-Based Verification Environment with XVC Manager

Block Interconnect Infrastructure Verification

A block interconnect infrastructure must support the peripherals, memory map, bandwidth and latency requirements of the system. Prior to integrating the system

peripherals with this bus interconnect infrastructure, it is desirable to verify the infrastructure as a design block in its own right.

A block interconnect infrastructure can be represented as a matrix or interconnect with a built-in arbitration algorithm as shown in Figure 8-5. It may also include additional supporting elements, such as bridging elements to other bus subsystems, data downsizers, data upsizers, bit-swizzlers and so on. These elements may also have been independently verified prior to integration into the bus interconnect infrastructure.

The block interconnect infrastructure verification environment is used to verify interconnect arbitration schemes and bus bandwidth requirements. It should also include a bus protocol functional coverage model. Transactions should be scoreboarded to verify the functionality of the bus infrastructure by tracing transactions from a master through the infrastructure elements to the target slave device.

When integrating combinations of interconnect infrastructure elements such as the ones mentioned above, it is desirable to ensure that, although they function correctly in isolation, they do not produce unexpected effects once combined. Diagnosing problems with interconnect elements and identifying the cause would be a difficult task using a fully populated system environment. Such an environment is not specifically designed to efficiently verify that connectivity. However, if a system-level verification environment is only concerned with checking the block interconnect infrastructure behavior, problems can be identified and diagnosed with greater efficiency.

Rule 8-35 — *All bus agents shall be replaced by XVCs.*

A bus agent is any design block that interfaces directly with the block interconnect infrastructure. A bus agent can be a master (proactive driver) or a slave (reactive driver). Unlike actual design blocks, XVC transactors can be controlled at a transaction or cycle level to create stimulus in patterns specifically designed to create system-level corner cases. If master or slave peripherals were used as bus agents instead, it would be difficult—if not impossible—to coerce them into creating the required stimulus to create the same corner case.

Recommendation 8-36 —*Constrained-random stimulus should be used.*

Constrained-random stimulus has proven to be the most effective method of verifying a block interconnect infrastructure. Even a simple infrastructure has an extremely large functional verification space due to the number of different transaction

permutations that it supports. Randomly generating stimulus will quickly achieve a high level of functional coverage and should hit many corner cases that are often overlooked when handcrafting directed stimulus.

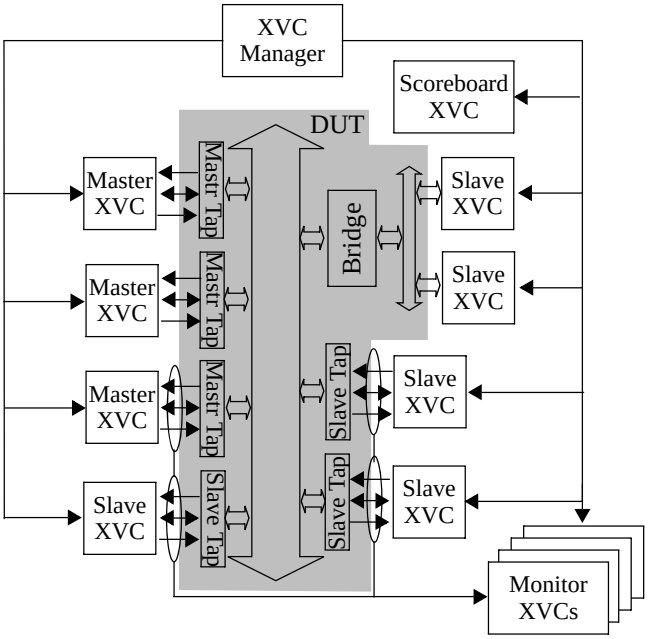


Figure 8-6. Structure of a Block Interconnect Infrastructure Environment

Figure 8-6 details the structure of a block interconnect infrastructure verification environment. It includes a bus interconnect and a single 64- to 32-bit downsizer element. Note the locations of the monitors in the environment and the presence of the scoreboard component that receives transaction information from each monitor via the XVC manager. Analysis of the resulting scoreboard coverage enables the verification engineer to modify the random stimulus to obtain the required functional coverage.

Figure 8-7 shows how XVCs in a block interconnect infrastructure verification environment are linked and communicate through an XVC manager. The XVC manager controls the overall simulation. Progress in the testbench is measured by monitoring the transactions that have passed through the interconnect infrastructure and by measuring the functional coverage that is being achieved. Upon completion of a test, the simulation is gracefully stopped by the XVC manager. Each XVC is

allowed to complete its current transaction—thus preventing false protocol or transaction monitoring errors.

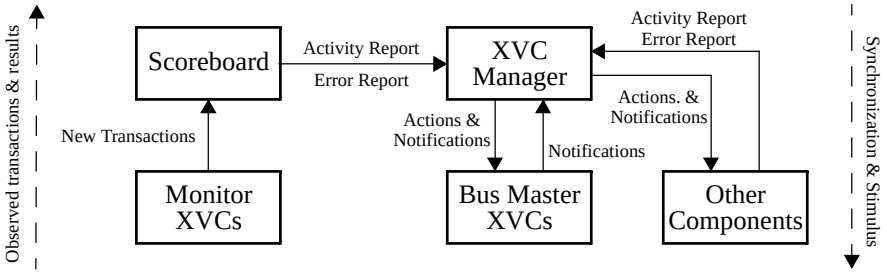


Figure 8-7. Simulation Control Communication in Random Block Interconnect Infrastructure Environment

Basic Integration Verification

The objective of basic integration verification is to verify the correctness of communications among design blocks.

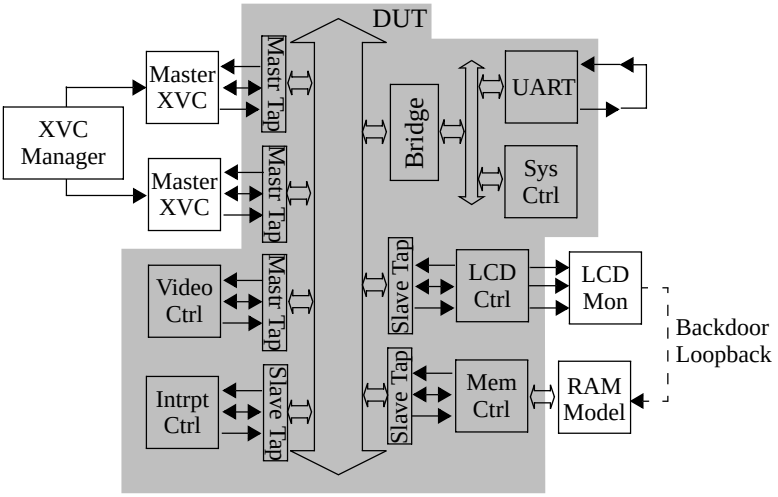


Figure 8-8. Structure of a Basic Integration Environment

Recommendation 8-37 —*The design blocks should be put into simple loop-back modes where possible.*

The loop-back approach allows an XVC bus master—usually replacing the CPU or DSP in the system—to be the sole source of stimulus in the integration environment.

This structure thus eliminates the need for synchronizing multiple stimulus streams and greatly simplifies the self-checking mechanism. For example, the external interface of the UART in Figure 8-8 is looped back. This loop back enables the UART control and status registers to be accessed and checked without requiring an external connection to the UART.

Care must be taken when using loop-back mode in certain peripherals such that the loop-back state does not hide a device configuration problem—i.e., where there is no inherent checking of correct device behavior. A possible solution, illustrated in Figure 8-8 by the LCD controller peripheral test block (PTB), would be to have a transactor that monitors DUT I/O and sends feedback to the master transactor indicating success or failure.

In some systems, it may not be possible to use a loop back structure on certain peripherals. For example, non-symmetrical devices require a corresponding master or slave component to enable data transfer. In this case, a verification component that provides the required interface functionality must be used.

Recommendation 8-38 —*Stimulus for software-accessible registers should be automatically generated.*

Stimulus for a bus master to access all software-accessible registers can be automatically generated from a set of configuration files that describe the locations and capabilities of software-accessible registers in peripherals. Recommendation 9-7 on page 351 briefly discusses register description. Figure 8-9 illustrates the process behind the generation of the master XVC's stimulus in Figure 8-8. One of the master XVCs uses the generated stimulus file.

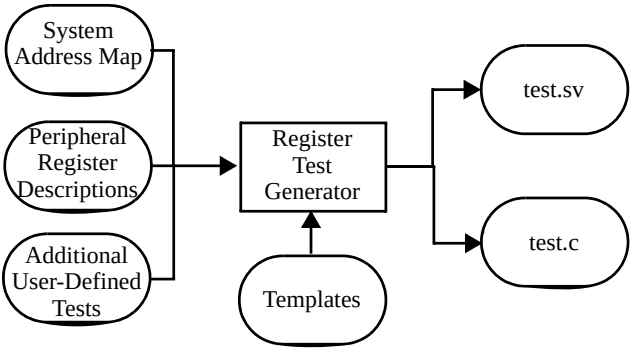


Figure 8-9. Automatic Generation of Register Integration Stimulus

The automatically generated stimulus provides a single-threaded basic register interconnect test for each peripheral in the system design. Automatic generation of the stimulus minimizes the overhead required when reconfiguring the system memory map or adding/removing peripherals. Note that the C source and C header files are used in the hardware/software verification environment as described in Chapter 9.

Recommendation 8-39 —*User-directed tests should supplement the generated register test.*

User-directed tests should be supplied to verify specific initialization and corner cases that cannot be automatically generated.

For example, a user-directed test would be required to verify that an interrupt line on a peripheral block has been correctly integrated into the interrupt controller block. The automatically generated register checks will establish that the interrupt controller status register and the peripheral configuration registers are accessible. But the user-supplied stimulus would be responsible for generating or forcing the interrupts and for polling the status register of the interrupt controller to determine that the correct interrupt is indicated.

Low-Level System Functional Verification

The previous system-level environments are useful for system integration verification, but do not readily cover the following potential functional verification requirements related to system integration testing:

- Reset mode checking
- Any control signals or configurations that cannot be determined or implied by a bus master agent read operation
- Any system integration functionality that is not looped back or visible or controllable by the CPU/DSP transactor

To address the above requirements, a low-level system functional verification environment builds upon the basic integration environment. At this stage, the system is being verified as a *known good* integration of *known good* blocks. As illustrated in Figure 8-10, the functional verification environment adds XVCs to provide external stimulus to the peripherals in the system.

Like the basic integration environment, the functional verification environment includes XVCs to drive and monitor all external peripheral interfaces. The CPU/DSP block continues to be replaced with a transactor to provide better control over the bus transactions they create.

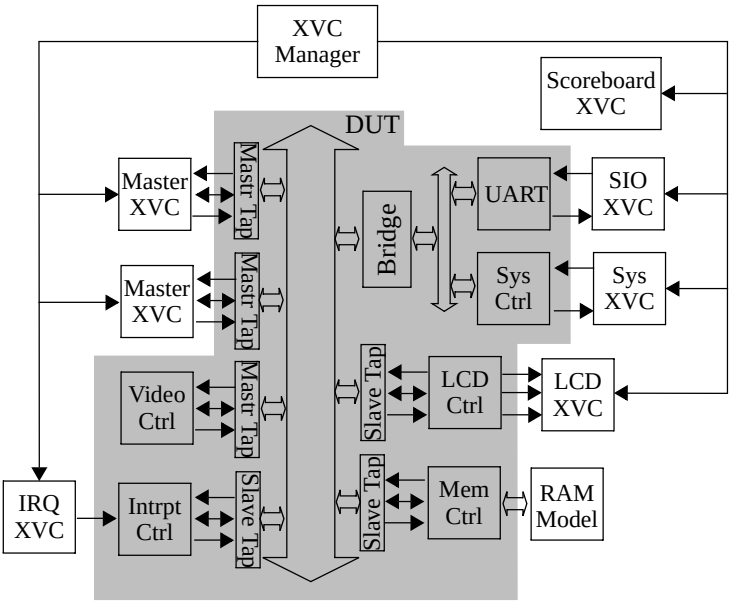


Figure 8-10. Structure of a Low-Level System Functional Verification Environment

System Validation Verification

The system validation environment concentrates on checking the performance requirements of the system. This environment features the pre-verified and integrated block interconnect infrastructure and design blocks. The maximum achievable performance of the block interconnect infrastructure can be verified using the interconnect infrastructure verification environment. The purpose of this environment is to ensure that the combination and interaction of the blocks still meets the required system performance requirements.

Figure 8-11 builds on the low-level system functional environment in Figure 8-10. It includes transaction monitors and scoreboard XVCs. In this example, the stimulus is targeted at verifying the bandwidth of the LCD controller master as it transfers image data from the memory model via DMA. Test results from the scoreboard can be correlated with the results from the LCD control XVC that checks the generated pixel data coming from the LCD controller itself.

.

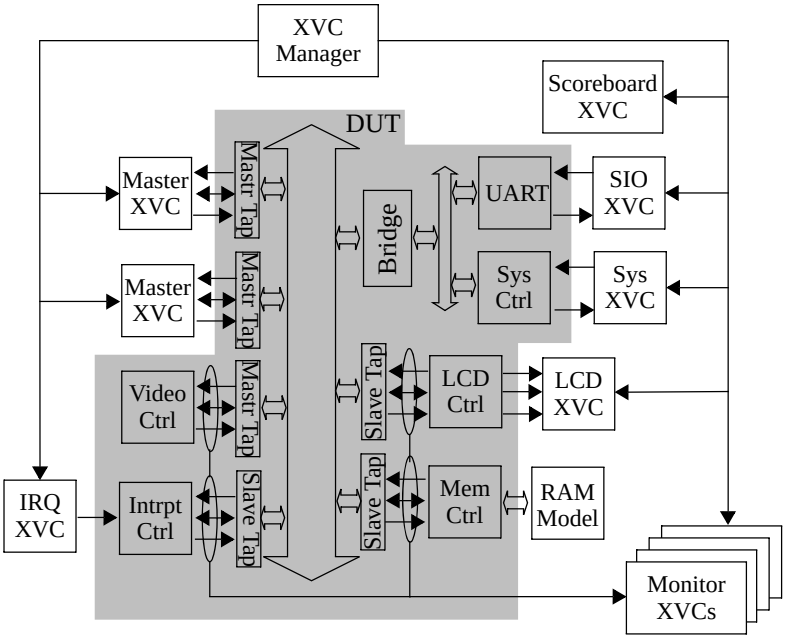


Figure 8-11. Structure of a System Validation Environment

Rule 8-40 — *The system shall be populated with models of the various blocks at the highest possible abstraction level suitable for the type of verification being done.*

Using RTL models for all blocks in the entire system usually yields a large system model that will be inefficient to simulate. Many verification requirements can be met with models at higher levels of abstraction. For example, for system-level performance analysis, it is often sufficient to populate the system using transaction-level models of the various design blocks.

Recommendation 8-41 —*The verification environments should be portable across abstraction levels of the system or its blocks.*

The system verification environment can be used to demonstrate that an RTL implementation is equivalent to the high-level transaction-level model of the system. By running the same test suite on different models of the system, they can be demonstrated to be functionally equivalent. “Verifying Transaction-Level Models” on page 332 expands on verifying transaction-level models.

Recommendation 8-42 —*The verification environments should be modular to optionally exclude selected verification components and/or design blocks*

Depending on the goal of a particular functional verification test, it may not be necessary to include all the XVCs for a particular test. It may also be possible to disable clock input signals to design blocks that are not associated with a particular test. Both of these techniques will aid in increasing the simulation or emulation performance and reduce test turnaround time. A modular verification environment using transactors that can easily be disabled—or not even instantiated in the `vmm_env::build()` method extension—should be considered.

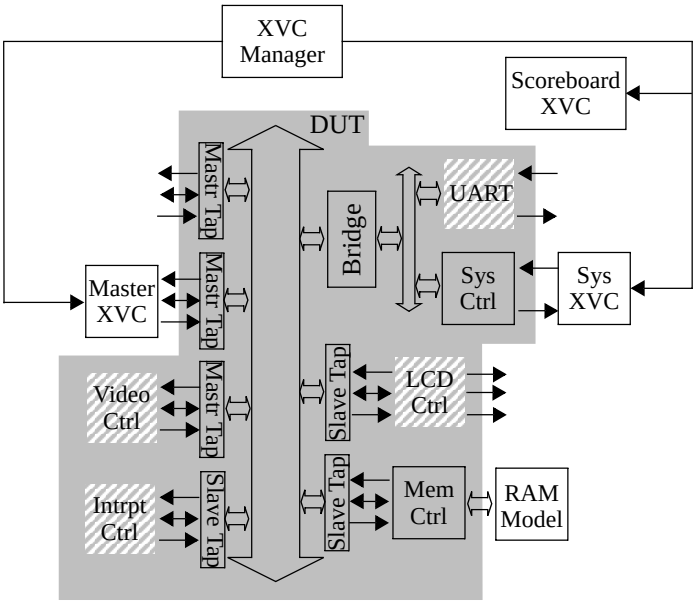


Figure 8-12. Verification Environment with Selected Blocks and Components Excluded

Figure 8-12 shows the verification environment from Figure 8-11 reconfigured to disable clock inputs to design and verification components not related to the functional test being performed. Disabled test components and peripherals are shown with hatched shading.

VERIFYING TRANSACTION-LEVEL MODELS

Design methodologies often include building a micro-architectural or transaction-level model of the design. This inclusion not only enables rapid prototyping and performance analysis, but also provides an executable specification for HDL implementation of the design. Like any model, the transaction-level model must be verified. A VMM-based verification environment can be developed to verify the transaction-level model.

Recommendation 8-43 —*Verification environments should be reusable across different abstraction views of the same DUT.*

If a verification environment is architected along layers, as described in "Testbench Architecture" on page 104, it can be reused from a transaction-level model to an implementation model, as illustrated in Figure 8-13. This reuse can shorten the time spent in RTL verification by writing tests at the transaction-level and by starting the development of these tests before the RTL implementation of the design is available. Of course, a test suite developed for a transaction-level model cannot always verify signal-level failures. The latter is better verified through additional tests or via formal techniques.

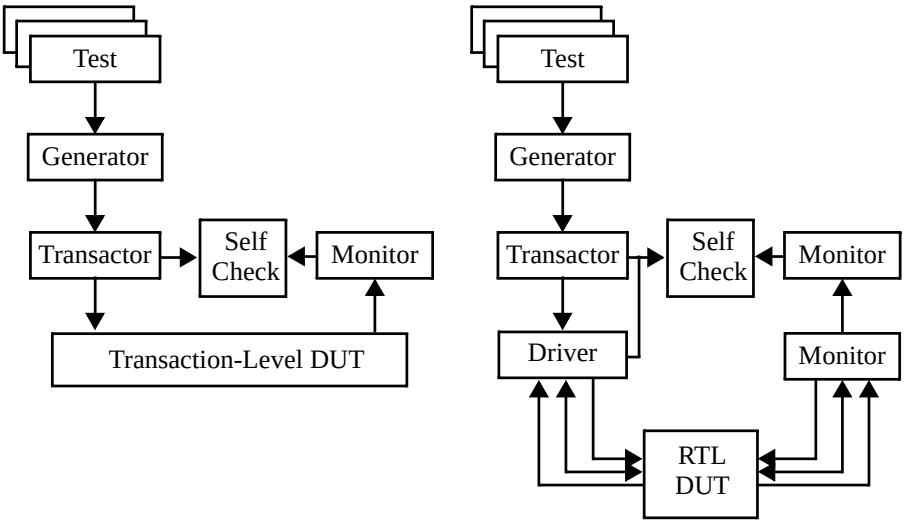


Figure 8-13. Reusing a Transaction-Level Verification Environment with an RTL Implementation

Recommendation 8-44 —*A transaction-level model of the design should be written.*

Writing a transaction-level model of the design is an intrinsic part of some design methodologies. For others, the first model of the design is written at the RTL level and writing a transaction-level model of the design appears to be a costly activity that will detract from the actual design. But writing a transaction-level model, if the RTL code is not yet available, is an investment—not a cost—that repays itself many times over the duration of the entire project.

A properly written transaction-level model only takes a fraction of the time it takes to write an RTL model, thus enabling the design and the verification team to work in parallel. It also executes orders of magnitude faster than the RTL model, allowing the verification environment and tests to be developed and debugged much faster. When the RTL model finally becomes available, all components of the verification environment are already at a very mature level. They can immediately verify the functional aspects of the design and can meet high functional coverage for these aspects.

Recommendation 8-45 —*A transaction-level model should have a pin-accurate shell.*

Transaction-level models are faster to write and simulate because they do not have to deal with the intricacies of physical signals and protocols. They can receive, execute and respond to transactions as high-level transaction descriptors. They do not need to exchange or operate on transactions as a collection or sequence of bits on some physical wire.

However, having the ability to substitute an RTL design block in a system with a pin-accurate transaction-level model will enable simulation to run much faster and with less demands on the simulation resources. By judiciously selecting a suitable mix of RTL and transaction-level models to populate a system, it is possible to meet the system-level verification requirements with a set of much lighter models. It may even be possible to meet all of the requirements without having to resort to emulation.

Rule 8-46 — *Transaction-level models shall not be written nor considered equivalent to RTL models.*

Transaction-level models and RTL models are not equivalent and should never be. A transaction-level model should be an abstraction of an RTL model. If it was equivalent to an RTL model, i.e., if the transaction-level model included all of the implementation details and artifacts, there would be no advantage in development and

simulation time. It is not possible to have a one-to-one comparison of a transaction-level model and a RTL model in all aspects—timing, delay, latency and behavior.

A transaction-level model—and the verification environment that verifies it—should only be as accurate as necessary, given a specification that describes timing and behavioral properties. The subsequent RTL model should be considered a subset of that specification.

Alternative 8-47 —*A transaction-level model can be written using SystemVerilog.*

Design methodologies that require or produce a transaction-level model of the design often use SystemC as the modeling language. It is an excellent choice and there are no reasons to use a different modeling language because the SystemC model may be used in other contexts than functional verification of the implementation. But for those design methodologies where a transaction-level model is written only to accelerate the development of verification environments, with no need to reuse that model in other context, SystemVerilog can also be a good modeling language choice.

SystemVerilog provides all of the necessary high-level constructs that make writing transaction-level models more efficient. Using SystemVerilog to implement a transaction-level model will also eliminate the technical challenges of integrating the SystemC model with the SystemVerilog verification environment. And as illustrated in Figure 8-14, verification components may also be used within a SystemVerilog transaction-level model to accelerate its implementation.

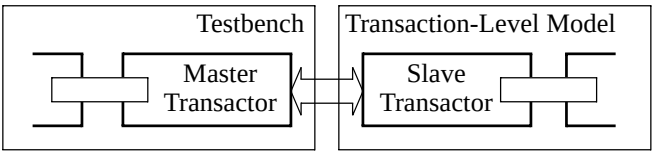


Figure 8-14. Using Verification Components to Write Transaction-Level Models

Transaction-Level Interface

Stimulus transactions must be transmitted from the verification environment to the transaction-level model. Similarly, response transactions must be observed from the transaction-level model by the verification environment. It is possible to exchange transactions via a pin-level interface but having to write a pin-accurate transaction-level model increases its complexity and its runtime burden. Transactions should be exchanged at a higher level of abstraction.

Rule 8-48 — *A `vmm_channel` shall be used to interface to the transaction-level model.*

The `vmm_channel` construct is a transaction-level interface used between transactors, as described in Chapter 4. The same mechanism can be used to exchange transaction descriptors between the verification environment and the transaction-level model, as shown in Figure 8-15.

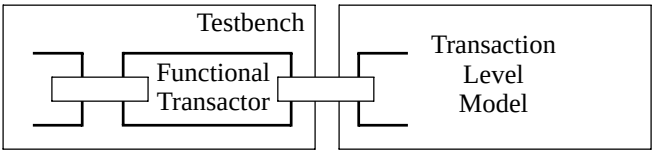


Figure 8-15. Transaction-Level Interface to a Transaction-Level Model

Using a `vmm_channel` to exchange transaction descriptors with a transaction-level model is simple to implement if the transaction-level model is written in SystemVerilog. But if a language boundary has to be crossed—for example, to verify a transaction-level model written in SystemC—the interfacing mechanism becomes more challenging.

Recommendation 8-49 —*Cross-language channels should be used if available.*

A standardized transaction-level communication mechanism offers opportunity for automation. Simulators that integrate SystemVerilog and SystemC can abstract the transition between the languages by providing channels with endpoints residing in the different languages. As illustrated in Figure 8-16, the upper layer of the verification environment would not be aware of the nature of the model at the other end of the channel. They can thus be reused, unmodified, from a transaction-level model to an RTL model. Cross-language channels automatically map transaction descriptors from one language into the other, without requiring both domains to have a common memory layout for their high-level data structures.

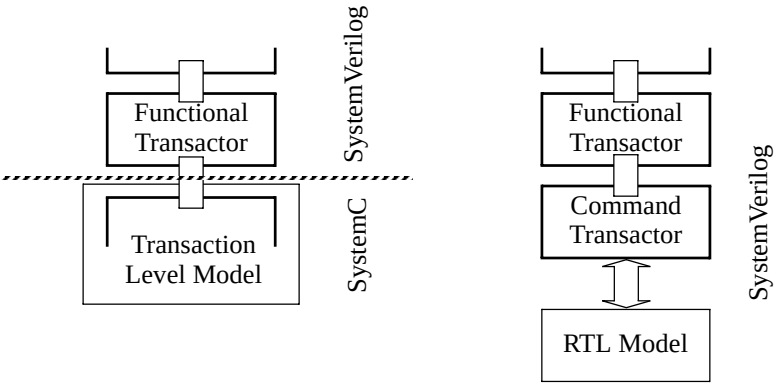


Figure 8-16. Portable Environments Through Cross-Language Channels

HARDWARE-ASSISTED VERIFICATION

In a fully simulated environment, a transactor can be used to drive the external interface of a peripheral. This is simple to accomplish because the peripheral, its external interface and the transactors are all included in the same simulation environment. However, there may be circumstances where the design must be verified on actual hardware. Emulation of the design is usually required because a pure software-based verification does not offer the runtime performance necessary to meet some verification requirements.

Peripherals integrated into a hardware-assisted verification of the design may have a number of interfaces to consider for verification. The bus interface is de-facto handled by the system itself. But any external interface needs to be driven or monitored during block- and system-level testing, and under the control of the simulation-based environment manager.

Although transactors are traditionally developed to interact directly with a simulated model of the design, it is possible to write a stand-alone, synthesizable RTL verification component for an external interface that has little or no dependency on a simulated verification components. Such a verification component, called a peripheral test block (PTB), could either operate standalone, could be controlled by an external driver such as a file-reader bus-functional model or could be controlled by an XVC.

Typical usage models for such a test block include:

- For FPGA implementation or emulation of the device or system under test
- For shipment as an out-of-the-box verification component to a customer with no particular dependency on the customer's verification tools
- For simple testbench environments where simple loop-back communications are all that are required for external interface testing of a device

Examples of peripherals that a PTB can address:

Sequential Data Peripherals — These peripherals transfer data sequentially, a single word, byte or nibble etc. at a time. The content or structure of the sequential data is not defined. Each transfer is independent of the previous or subsequent transfers. Peripherals of this type include UARTs and GPIO.

Sequential Block Peripherals — These peripherals transfer data sequentially, a block of data at a time (multiple words, bytes or nibbles etc.). The format of the data in the block is defined for individual transfers and blocks are transferred in a fixed order. Peripherals of this type include video controllers and network interfaces (Ethernet, USB etc.).

Random Block Peripherals — These peripherals transfer data a block at a time. The format of the data in the block is defined for individual transfers and blocks are transferred in an arbitrary order. Peripherals of this type include Serial Peripheral Interface, Memory Stick, SD-Card and Multi-Media-Card reader interfaces.

Examples of peripherals that a PTB does not attempt to address include:

Random Data Peripherals — These devices transfer the data in a random way for example a single word, byte or nibble etc. at a time. The format for the data is not defined for the transfer larger than the entity size and data can be read out in any order. Peripherals of this type include ROMs and RAMs.

Rule 8-50 — *A peripheral test block must have a generic external interface.*

The PTB is a way of creating a synthesizable transactor for peripherals within an emulated system design. It can be reused between different hardware-based verification environments. It should therefore present a generic external interface to suit different integration mechanisms, such as direct interaction with transaction-level transactors or through software-programmable register interface provided by a hardware wrapper.

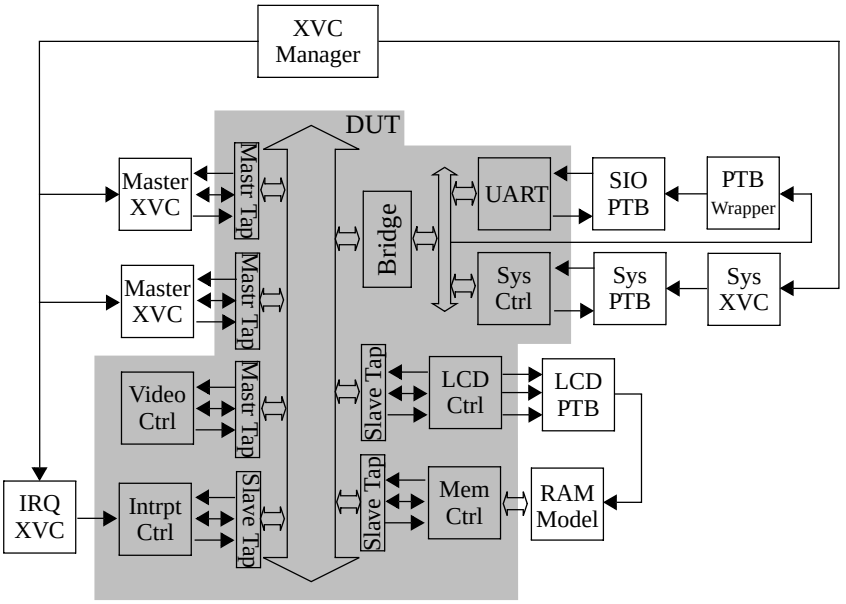


Figure 8-17. External Interface on a Peripheral Test Block

Figure 8-17 illustrates the external interfaces that a PTB can have. In this example, the PTB for the UART has an AMBA APB interface wrapper to enable it to be set up and controlled by the verification environment. Also in this example, the PTB for the system controller peripheral is directly controlled by an XVC. The XVC may be used in a simulation environment to perform random functional testing on the system controller, and the XVC may generate a set of test vectors that can be replayed at FPGA level or at silicon level. The PTB wrapper may be an external AMBA APB interface wrapper as used on the UART device. The LCD PTB does not require XVC interaction or external control in this example, and the PTB can be reused from simulation through to FPGA and silicon testbench environments.

Peripheral Test Block Structure

Figure 8-18 illustrates the internal structure of the PTB. The interface to the DUT is on the left of the diagram, and the control interface is on the right of the diagram.

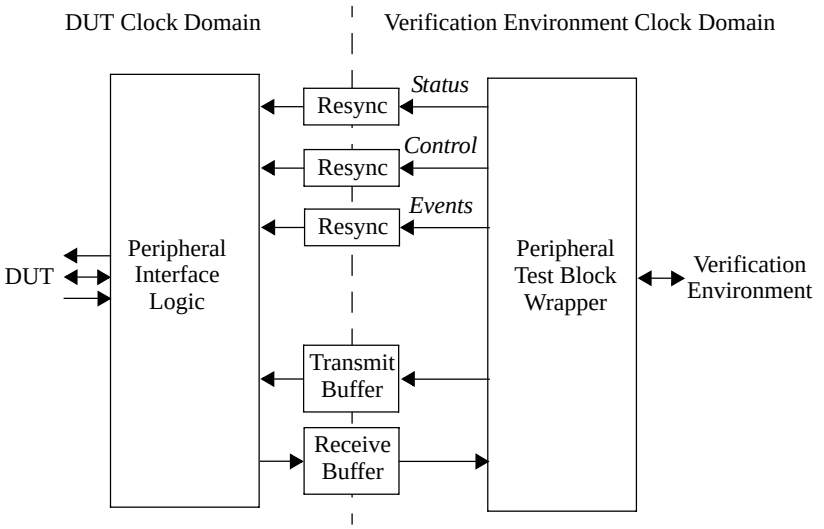


Figure 8-18. Structure of Peripheral Test Block

Rule 8-51 — *A peripheral test block shall have a peripheral interface logic block.*

A PTB has a physical interface layer similar to the command-layer transactor of an XVC that communicates directly with the physical-level interface of a DUT. This block works in the peripheral clock domain and performs data translation at the DUT interface. Protocol checking of the data and associated signals can also be done using SystemVerilog-based assertions. One such example would be to use the synthesizable subset of the VMM checker library described in Appendix B.

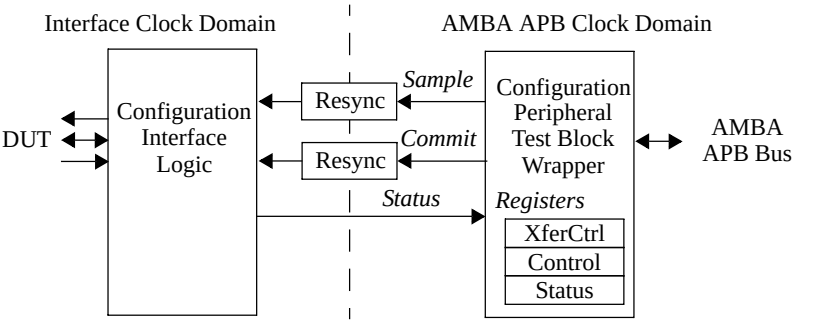


Figure 8-19. Interfacing to a Peripheral Test Block

Rule 8-52 — *Transferring non-static data across clock domains shall require synchronization and acknowledgement.*

Data that crosses asynchronous clock domains must be synchronized to the new clock domain using two or more flip-flops. Because of the asynchronous nature of the resynchronization, there is no way for the upstream clock domain to know when all the bits of the data have been successfully transferred to the downstream clock domain. Providing an acknowledge signal back into the upstream clock domain explicitly confirms the successful transfer and the possibility of transferring another datum.

In Figure 8-19, the `Config` peripheral interface provides a `Commit` and `Sample` register in the peripheral clock domain and returns an acknowledge in the DUT clock domain for commit and sample operations to indicate that the action is completed. A synchronized transfer is initiated by writing data in the `Transfer` register with the appropriate `commit` or `sample` bit asserted. The completion of the transfer can be detected by reading the `Transfer` register with the `CommitAck` or `SampleAck` bit asserted.

Recommendation 8-53 —*A peripheral test block should support control registers.*

Control registers should be available in the PTB to be accessed from the peripheral test interface as *n*-bit-wide registers. Control registers act as general-purpose configuration registers for the DUT and are the hardware communication input channels for the peripheral interface logic. For example, a UART PTB designed to test a UART would have control registers to program the baud rate, parity stop bits, buffer depth and so on.

Each of these registers can be written in any order without initiating an immediate transfer. The contents of all control registers are transferred in a single cycle onto the control bus to maximize simulation/emulation efficiency between the peripheral test interface and its driver.

In a simulation-based environment, the control signals are directly driven by an XVC. The XVC can modify all bits of the control signals in a single access. This ability removes the requirements of a hardware peripheral test interface.

Recommendation 8-54 —*A peripheral test block should support status registers.*

Status registers should be available in the PTB to be accessed from the peripheral test interface as *n*-bit-wide registers. Sample registers act as general-purpose feedback registers from the DUT and are the hardware communication output channels for the

peripheral interface logic. Each of these registers can be read in any order without initiating an immediate transfer. The contents of all sample registers are transferred in a single cycle from the status bus to maximize simulation/emulation efficiency between the peripheral test interface in the same manner as their corresponding control registers.

In a simulation-based environment, the status signals are directly sampled by an XVC. The XVC can sample all bits of the status signals in a single access. This ability removes the requirements of a hardware peripheral test interface.

Recommendation 8-55 —*A peripheral test block should support event signals.*

Event signals should be available in the PTB. Event signals are intended as general-purpose asynchronous feedback notification signals. Typical usage includes monitoring transfer buffers, indicating DUT error conditions or assertion completions. Each event is a single signal so no specific requirement is placed on making sure some signals do not change before others.

If a transfer of information greater than a single bit is required, then the information should be included in the status signals and an event signal used to indicate that information is available. When the event is observed, the peripheral test interface can read the status signals (as defined in Recommendation 8-54) to recover the required information.

In a simulation-based environment, the event signal is directly accessible from an XVC. The XVC can monitor it directly. This ability removes the requirements of a hardware peripheral test interface.

Recommendation 8-56 —*A peripheral test block should support a transmit buffer for hardware-only environments.*

A transmit/write buffer should be available in the PTB to support buffered asynchronous transfers between the two clock domains. The data is transferred on a first-in, first-out basis and a control flow mechanism should ensure that no data is lost due to overrun. A transmit buffer may be preferred over control registers for burst data transfers.

In a simulation-based environment, there is no requirement for such a buffer: The XVC used as the peripheral test interface has the ability to transfer data in zero simulation time.

Recommendation 8-57 —*A peripheral test block should support a receive buffer for hardware-only test environments.*

A receive/read buffer should be available in the PTB to support asynchronous transfers between the two clock domains. The data is transferred on a first-in, first-out basis and a control flow mechanism should ensure that no data is lost due to overrun. A receive buffer may be preferred over status registers for burst data transfers.

In a simulation-based environment, there is no requirement for such a buffer: The XVC used as the peripheral test interface has the ability to transfer data in zero simulation time.

SUMMARY

Along with being reusable, any XVC or verification environment must be flexible enough to suit the given system under test, e.g., bus width, protocol subset restrictions and generation constraints. All must be easily configurable via parameters.

The following functionality should be provided by reusable verification components:

- Directed stimulus generation
- Constrained-random stimulus generation
- Block-level synchronization
- System-level synchronization
- Protocol integrity checking
- Error injection
- Functional coverage data collection

An XVC manager component is used to synchronize and schedule the above functionality. This single control point also simplifies the development and maintenance of tests.

Finally, when migrating the design to abstraction levels where high-level verification components are not readily supported, hardware-assisted verification using a PTB may be required.

CHAPTER 9 PROCESSOR INTEGRATION VERIFICATION

Verification using embedded software is an important part of any system verification infrastructure where a host processor directs application data and controls memories and peripherals. For system-level testing where a CPU or DSP is part of the system design being tested, it is desirable to have a verification environment that supports the execution of test software to demonstrate that the system can successfully support the execution of an operating system, application software or a DSP control algorithm. This chapter, and its companion Appendix D, will be of interest to block- and system-level architects and verification engineers, with a focus on system-level integration verification using software.

This chapter introduces the concept of software test environments to complement the hardware-centric infrastructure described in the previous chapters. The environment is used in place of an operating system in a CPU-centric system design. System verification can thus be conducted prior to the operating system being available on the system itself. XVCs in the verification environment will work in concert with the software test framework such that both external and software-internal stimulus can be generated and synchronized to create interesting and relevant system conditions to meet the hardware/software verification requirements.

SOFTWARE TEST ENVIRONMENTS

Recommendation 9-1 — *CPU or DSP integration should be tested using software test routines.*

Where the system design includes a CPU or DSP, software test routines must be used to direct the processor(s) to address the system peripherals they are responsible for interacting with. This direction is needed because the bus-functional model's approach, although good for integration testing, often will not have the same timing characteristics as the end processor to be used.

For example, a processor may insert an extra idle cycle due to an internal processing operation. The arbitration unit may decide to take away bus ownership during that extra idle cycle and hence delay a slave transaction or even defer it to another master. In contrast, a bus-functional model may generate slightly different behavior whilst still driving the bus with the same stimulus. Were the bus-functional model not to generate an additional idle cycle in this scenario, the arbitration handover would not take place, leading to a different scenario being played in the system.

In another aspect, unless the CPU/DSP transactor also emulates the instruction fetch-and-execute cycle from external memory, it is also unlikely to exhibit the same behavior as the actual CPU or DSP in operation. It would be possible to program the transactor to exhibit this behavior, but in practice it is far simpler to program the CPU or DSP to execute instructions to perform this task.

Chapter 8 introduced different hardware-centric, system-level verification environments. For software verification of systems, two additional environments, illustrated in Figure 9-1, are required.

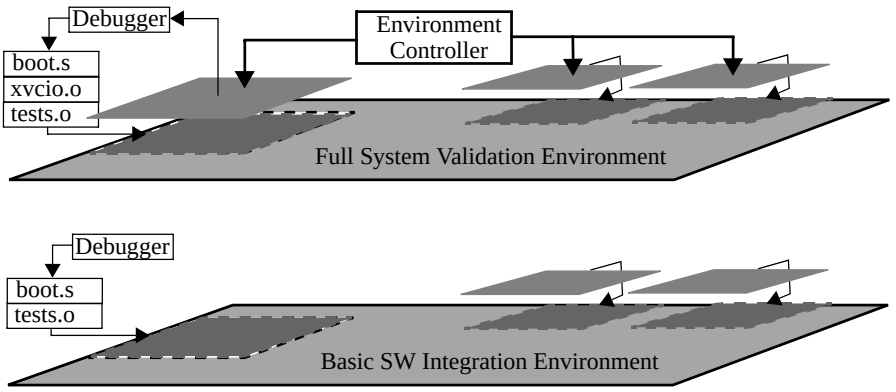


Figure 9-1. Processor Integration Verification Environments

Verification can proceed at any system abstraction level. However, a system design implemented at a given abstraction level may be more or less suited to one verification environment over another.

Basic Software Integration Environment — To support Recommendation 9-1, this environment supports integration verification using the actual CPU/DSP. It is assumed that basic integration testing has been performed on the system such that the block interconnect infrastructure and integrated design blocks provide a *known good* starting point. The basic software integration environment also supports a limited degree of peripheral design block integration verification.

Full System Environment — This environment enables full functional verification, such as OS booting and execution of application code and/or hardware benchmarking tests. A combination of code and verification components is used to provide system stimulus, usually with the verification components providing stimulus on the external interfaces of the system. This environment should be used with an abstraction model of the system capable of high simulation/emulation speed.

Basic Software Integration Verification

The primary role of this environment, when compared to “Basic Integration Verification” on page 326, is to ensure that the system CPU/DSP is correctly integrated. This integration is achieved by executing software test routines on the CPU/DSP such that it interacts with the rest of the system peripherals.

Recommendation 9-2 — *Design block integration verification should be performed using this environment as required.*

Using software is a more indirect approach to integration verification as it demonstrates integration by stimulating peripheral registers or stimulates a device's external interface via a memory-mapped verification component. This type of testing is more accurate in nature and verifies the timing and protocol of signals between the CPU and the devices to which it is directly coupled.

Recommendation 9-3 — *The basic software integration verification environment should support a single software threading model.*

Single-threaded software verification environments make it very simple to run and debug multiple tests. Verifying correct interaction among different system peripherals is also straightforward to achieve using a single-software thread where the number of interacting peripherals is constrained to a small number. For example, a software

control thread can be used to program timer and interrupt controller peripherals first, then monitor transmission of data through a serial port device next.

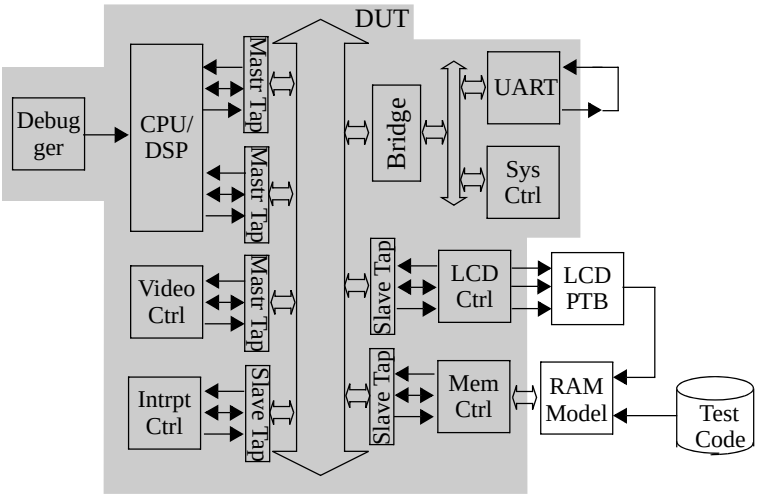


Figure 9-2. Structure of a Basic Software Integration Environment

Figure 9-2 shows an example of a basic software integration verification environment. A software debugger may be used to trace the progress of the test code, which is loaded as a binary image into the environment memory model. When a debugger is used, it is also possible to load an alternative code image via the debugger.

In this example environment, complex interaction among peripherals is not required. Rather than using XVCs, a single PTB-based monitor is used to confirm that the LCD controller has been initialized correctly by the software (see “Peripheral Test Block Structure” on page 339 for more details on PTBs). The primary goals for this example are to check that the CPU can access each of the system peripherals program registers and that it can successfully load and execute instructions using the memory controller. Other than performing register tests, a simple scenario of initializing the LCD controller to output a sample image is also included in the software test code.

Full System Verification Environment

Figure 9-3 shows a system verification environment that builds upon the hardware integration environments described in Chapter 8 and the basic software integration environment described earlier in this chapter. This environment should only be used

once a high level of confidence in the integration of the design blocks has been achieved.

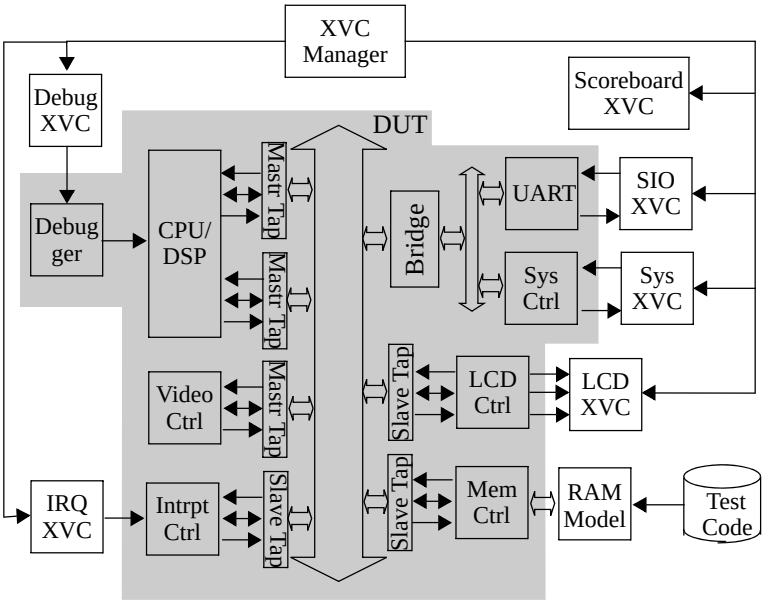


Figure 9-3. Structure of a Full System Verification Environment

There are two considerations when verifying functionality using this software-based verification environment:

1. The visibility and controllability over system-level hardware events is more limited than the hardware-oriented system-level testbench environments described in Chapter 8.
2. It is not possible to verify the operation of bus transactions outside of the range of bus transaction types supported by the host processor model used in the environment—a consideration when the system is to be designed for compatibility with different CPU types.

Despite these considerations, this verification environment can be very effective in furthering system-level verification to include:

- Prototyping high-level software driver functionality
- Supporting an OS boot given an execution environment that can run the design between hundreds of KHz to several MHz or faster
- Complex interaction scenarios such as simultaneous interrupt and DMA requests handling from multiple peripherals

- In-system functional testing of peripherals by playing them off against each other, e.g., multiple bus masters accessing the same memory controller

XVCs and an XVC manager can be used to interact with the system test software to address the visibility and controllability issue.

Suggestion 9-4 — *The software test framework can communicate with the XVC manager.*

The XVC manager can be connected to the software execution environment via a debugger. This connection lets certain test scenarios be controlled from the XVC manager instead of the main software execution thread. It allows the coordination of hardware events or protocol being monitored by any of the XVCs present in the system with software actions being executed on the CPU.

Suggestion 9-5 — *The full system verification environment can support a multi-threading model.*

A single-thread model is useful for the reasons described in Recommendation 9-3, but it does not offer accurate timing of cross-peripheral interactions. Transmitting data concurrently through multiple serial ports would require a far more complex and difficult to debug single-threaded control program. By letting users execute multi threaded tests, it is possible to specify concurrent tests as if they were individual single-threaded tests. Multi threaded tests can thus emulate operations such as:

- Concurrent interaction with two different peripherals
- Multiple peripherals concurrently transferring data across a system
- Concurrent or real-time operating system services

Concurrent test execution can be implemented by having a scheduler maintain a set of threads, one per test. Software tests can be written and run as if they were in a single-threaded environment, but a new thread can be created at any time to run another test concurrently. In addition to threading on one processor, support for multi-processor threading may be a requirement of a software verification environment, depending on the configuration of the system being tested.

Using a multithreading software verification model may be considered a lower priority requirement when a block interconnect environment is used to fully verify latency and bandwidth prior to system integration, and an OS boot test is executing multithreaded software tasks to access system peripherals.

Recommendation 9-6 — *Verification of a CPU-based system should include an OS boot.*

During an OS-boot sequence, the system exercises interaction sequences among design components in a way that is representative of its real-world operation. It is a form of verification that would otherwise be impractical to set up with the any of system-level verification environments mentioned in Chapter 8.

A typical embedded operating system, such as embedded Linux, may require in excess of one billion cycles to complete its boot sequence. For practical reasons this requirement can only be met on a transaction-level model of the system or a hardware-assisted verification environment due to the sheer number of cycles required. An RTL design in an emulated environment can execute at up to 400 KHz. The same RTL design in a pure software simulation environment may only execute at up to 1 Khz. Thus, a software test requiring 36 million cycles to complete would take approximately 10 hours of real time to execute in the pure software simulation environment. The same test run on the emulation environment would be expected to complete in approximately 90 seconds.

In contrast, depending on the changes made to the RTL between tests, it can take up to an hour to completely rebuild and remap the design to the emulator. To compile and rerun the same code for execution on the software simulator requires only a few minutes.

The net gain between using an emulator and a software simulator can be argued as marginal in the early phases of a project. At this stage, the design is still being debugged and modified frequently and the simulation tests can be partitioned to run in under an hour. However, the same RTL design would be expected to complete an OS boot in a software simulation environment in approximately 12 days, compared to just 40 minutes on an emulator. A complex test, such as an OS boot, is only attempted when the design is mature and believed to be functionally correct. The few rebuild and re-mapping cycles that may be necessary to complete the test in this case would not outweigh the gain in runtime performance over software-based simulation.

STRUCTURE OF SOFTWARE TESTS

The extensibility of the software verification environment must be as straightforward as possible when new peripherals are added or new tests designed. The following guidelines will help in implementing a modular code structure sufficient for supporting a wide range of verification challenges in a bus-based system.

Figure 9-4 is an overview of the software verification framework structure described in this chapter. Note that the automated test generation engine introduced in Chapter 8 is being reused here to generate peripheral register description header files.

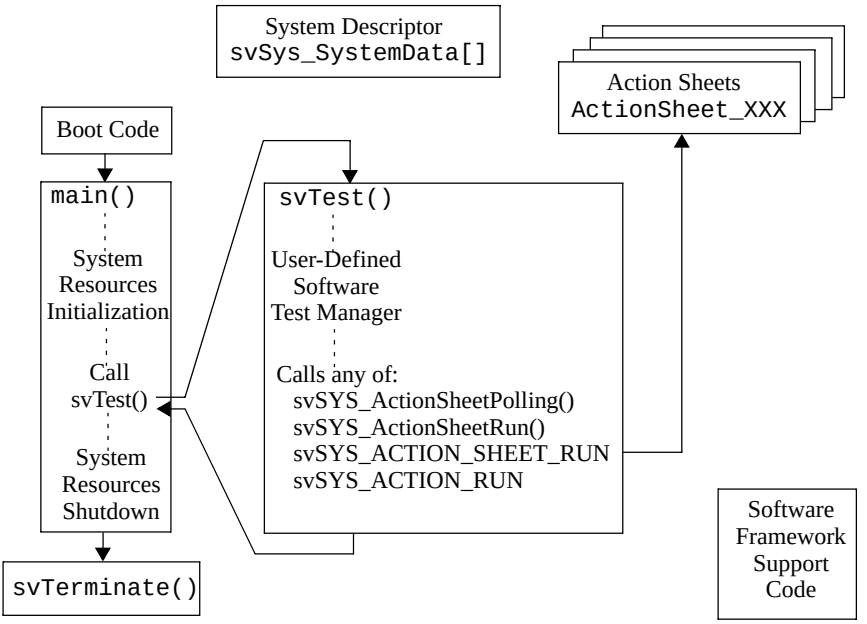


Figure 9-4. Software Verification Framework Structure

Figure 9-5 shows a left-to-right flow of how software tests are compiled and executed on the software verification environment above. Refer to Appendix D for more detail on software verification framework structure definitions.

.

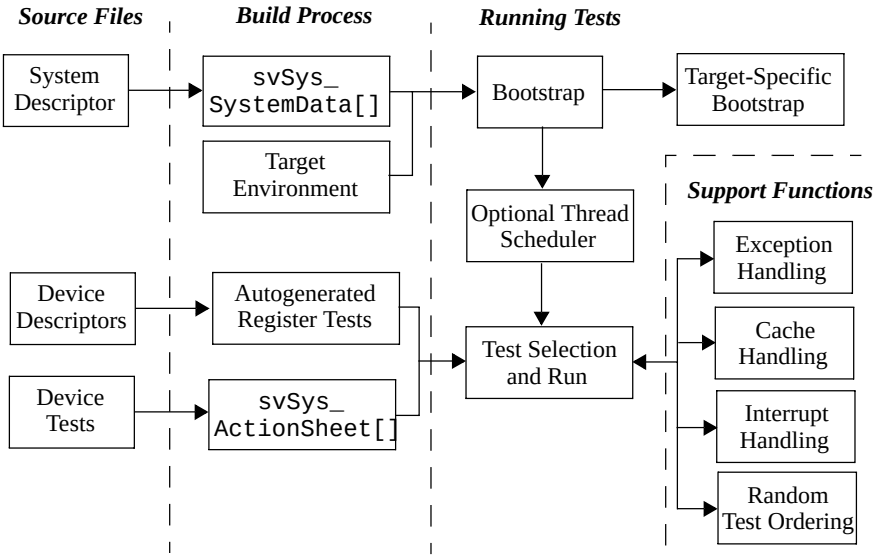


Figure 9-5. Software Test Execution Process

Recommendation 9-7 — *All software-accessible peripherals should be described in a system descriptor.*

It is recommended that a system descriptor be available for the software verification environment. This system descriptor contains information on each software-accessible system peripheral. The information it contains is based on the design specification such as system memory map, interrupt and DMA request line allocations. The information can be extracted from the design or specification document, or portions of the design can be automatically generated from the system description as shown in Figure 9-5 above.

The example below shows an XML description of a single software-accessible register in a peripheral. It describes an interrupt clear register in the peripheral named `GPIO_1` residing at system address `0x30000000`.

Example 9-1. Register Description File

```
<peripheral name="GPIO_1" offset="0x30000000">
   ...
   <register name="IC" offset="041C">
      <descriptivename>Interrupt clear register</
descriptivename>
       <description></description>
       <bitfield name="IntClear">
          <function></function>
          <bitpos>0</bitpos>
          <size>8</size>
          <initialvalue>0</initialvalue>
          <accesstype>RO</accesstype>
       </bitfield>
   </register>
   ...
</peripheral>
```

Suggestion 9-8 — *Elements of the system software can be auto-generated from the system descriptor.*

This system descriptor can be turned into C code that is directly usable by the system software. For example, the C code in Example 9-2 can be generated from the (partial) system descriptor Example 9-1.

Example 9-2. Automatically Generated Register Test C Code

```
volatile struct s_IC {
   unsigned IntClear : 8;
};
typedef volatile struct {
   union {
      ...
      union {
         s_IC IC;         // Interrupt clear register
         unsigned IC_raw; // for raw access };
         ...
      }
   }
} t_system_peripheral_GPIO_1;
```

Recommendation 9-9 — *The system descriptor should be translated into an array of type* `svSYS_SystemElement` *named* `svSYS_SystemData`*.*

The predefined *svSYS_SystemElement* structure contains all the relevant information required by the software verification environment support code to access peripherals. See Appendix D for more details.

The values of each field are automatically generated by the build process. Where system description fields cannot be readily determined from the system description data, default values are used.

Recommendation 9-10 —*Symbols should be used to index into the system descriptor data.*

As peripherals are added or removed from the system, the location of peripheral descriptors in the system descriptor may be modified. To avoid breaking existing tests, no peripheral descriptor should be accessed using a hardcoded location in the system descriptor. If symbolic values are used, only the value of the symbols needs to be updated.

Recommendation 9-11 —*Enumerals named* *svSYS_Element_XXXX_N* *should provide symbols to index into the system descriptor.*

A peripheral named *XXXX* may be instantiated more than once. Each instance is differentiated using an instance number *N*. For example, the symbols to access the descriptor for instance #0 of peripheral P123 is named *svSYS_Element_P123_0*. The enumerals must be defined in the same order as the peripheral descriptors they correspond to in the system descriptor.

Example 9-3. Symbols to Index into the System Descriptor

```
typedef enum {
   ...
   svSYS_Element_P123_0, //Symbolic index P123 #0
   svSYS_Element_P123_1, //Symbolic index P123 #1
   ...
}
```

Recommendation 9-12 —*All externally visible identifiers should follow the naming convention.*

Externally visible identifiers may conflict with each other if they are not kept separate.

This chapter and Appendix D uses the prefix *svMOD_* for all externally visible identifiers, where *MOD* is a unique identifier for the code module the identifier

belongs to. The module identifier `SYS` is reserved for the support software. `sv` stands for software verification, not SystemVerilog.

Recommendation 9-13 —*Each system descriptor should have a corresponding test action sheet.*

An action sheet is used to group all of the associated test actions with a particular peripheral. An action sheet item comprises:

- A text description for the test actions name
- A test level for the test action to be used with the test level specified in the peripherals system descriptor
- A pointer to the test action function

Along with user-defined test actions, the build process generates two default tests based on the description of the system peripherals:

- An initial state test: Designed to check the successful initialization of the peripheral after system reset
- A register test: Designed to check the accessibility of each of the programmer's registers on the peripheral

TEST ACTIONS

The guidelines presented in this section will help define and implement software test actions.

Recommendation 9-14 —*Device tests should be in the form of test actions.*

For every peripheral, there shall be one or more test actions. Each action is designed to test a specific part of the peripheral's functionality. These actions are written predominantly in C, with assembler being used where necessary. Every action should be self verifying by checking the result of the action against an expected result and return a pass or a fail indication. The software verification framework will output a message should the action fail.

Recommendation 9-15 —*Test actions source files should be reusable by the software build process from one system design to another.*

Where reusable peripheral blocks in a bus-based system may have corresponding reusable verification components, it is also possible to treat corresponding test actions

as reusable verification IP in the same context. A test action is built with a common method name/entry point and parameter list, and extracts its operating parameters from dynamic data held within the parameter list rather than from a set of predefined system constants. The dynamic data itself is generated from a system description.

Recommendation 9-16 —*Actions should follow a naming convention.*

A general object code linker requires that any build configuration cannot contain functions with common names.

Furthermore, the build system should be able to automatically generate a test software framework for any given system design with a default set of test actions. This automatic generation requires that it can parse explicit test actions from within the source code directory structure. In this case, action *YYYY* for device *XXXX* must be named `svXXXX_YYYY` such that the build process can identify it as a valid test action.

Recommendation 9-17 —*Actions should adhere to the* `svSYS_SeqTest()` *function prototype.*

For the same reason that test actions must follow a set naming convention for the bootstrap module to be able to call arbitrary actions, test actions must also follow a predefined prototype.

Example 9-4. Defining a Software Test Action

```
svSYS_eTestResponse
svP123_FirstAction(svSYS_SystemElement * SysData)
{
   ...
}
```

Recommendation 9-18 —*Each test action should define a name by using the* `svTEST_NAME()` *macro.*

This macro associates an arbitrary text with a known symbol in the action object file. This association is so that test actions can be called in a uniform manner whilst providing good debug visibility.

For example, the `svP123_FirstAction` action would define a name as:

```
svTEST_NAME(svP123_FirstAction, “Test Interrupts”)
```

Recommendation 9-19 —*Each test action should define a test complexity level by using the `svTEST_LEVEL()` macro.*

This macro associates an arbitrary integer with a known symbol in the action object file. This data is added to the system descriptor data to let the user select groups of tests appropriate to the test level that is required, i.e., register tests, functional testing, peripheral interaction and so on.

For example, the *`svP123_FirstAction`* action would define a level as:

```
svTEST_LEVEL(svP123_FirstAction, 3)
```

Recommendation 9-20 —*A test complexity level should be no greater than 31.*

The test runtime environment provides a mechanism for selecting tests based on their complexity level using bits in a 32-bit value. This limits the complexity level to values in the 0 to 31 range inclusively.

Recommendation 9-21 —*Action should use the `svSYS_GET_SYS_DATA()` macro to access other peripherals.*

Actions that target a single peripheral only require the information supplied via their *`SysData`* argument. But if an action requires access to other peripherals, that action needs a mechanism to access the descriptors for those peripherals. This access is accomplished by the *`svSYS_GET_SYS_DATA()`* macro. See Appendix D for more details.

Recommendation 9-22 —*Any reference to data outside of the system descriptor should be enclosed by the `svSYS_HARDCODED()` macro.*

In rare occasions, it may be necessary to access test data that is not available from the system descriptor structure. This need for access will affect portability of the tests. The software framework offers the *`svSYS_HARDCODED()`* macro to identify these cases.

Recommendation 9-23 —*The `svIO_BYTE_READ()`, `svIO_BYTE_WRITE()`, `svIO_WORD_READ()` and `svIO_WORD_WRITE()` should be used to access memory-mapped registers in peripherals.*

These macros are provided by the software framework to access absolute memory locations in the data space.

Recommendation 9-24 — *Exceptions should be thrown by calling the* `svSYS_ThrowException()` *function.*

Any action may throw an exception. Exceptions may be thrown when actions wish to terminate when an error condition is detected in a subroutine. Exceptions are thrown and caught in a system-specific way. The `svSYS_ThrowException()` `function` hides the system-specific mechanism for generating a processor interrupt and exception handling.

Example 9-5. Throwing an Exception on a Software Test Action

```
if (...) svSYS_ThrowException();
```

Suggestion 9-25 — *Assumptions can be checked using the* `svSYS_ASSERT()` *macro.*

Any assumption or expectation made in a software test should be verified using an assertion. The assertion will throw an exception if it fails.

Example 9-6. Checking an Assumption

```
svSYS_ASSERT(Mode == 0);
```

Suggestion 9-26 — *The* `svSYS_CACHE_BLOCK_START()` *and* `svSYS_CACHE_BLOCK_END()` *macros can be used to define code regions to be locked in the instruction cache.*

These macros provide an easy method to identify the boundaries of cacheable code regions. They mark the start and end of a block by introducing an empty function. It is expected that entire test actions will be cached.

Example 9-7. Identifying the Boundaries of Test Actions

```
svSYS_CACHE_BLOCK_START(MyBlock)

svSYS_eTestResponse P123_FirstAction(svSYS_SystemElement
* SysData)
{
   svSYS_CacheLines locked =
   svSYS_CacheLock(svInstructionCache,
                   &MyBlock_CacheBlockStart,
                   &MyBlock_CacheBlockEnd);
   ...
   svSYS_CacheUnlock(locked);
}

svSYS_CACHE_BLOCK_END(MyBlock)
```

Because functions are used to create the boundary symbols, it is not possible to lock individual code statements in the cache. Only entire functions can be locked down. Note that no such macro exists for the data cache.

Suggestion 9-27 — *A register test action can be generated from the system descriptor.*

For every peripheral, a basic register test action can be generated from the system descriptor. The test action should only use the `pSysData->BaseAddress` data. The test should check that the base address lies in an uncached and unbuffered region of the memory management unit page table and should exit with `svTestFailed` otherwise. The test should then proceed to verify that all bits in the software-accessible registers in the peripheral are accessible and can be written or are read-only.

Note that not all registers may be automatically testable in all peripherals. If the behavior of certain registers vary depending on the value of other registers, it will not be possible to automate the verification of those registers.

Recommendation 9-28 — *Test actions should use `svSYS_Printf()` function to issue all messages.*

The message service described in “Message Service” on page 134 may be available to software tests. Using a single function to issue all messages lets them be directed to the message service should it be available, or provide similar functionality.

Note that the messages that can be issued from a software test are much simpler than in a simulation-based environment. The message may have to be produced and saved in a hardware-based platform that does not have the same I/O capability as the simulation-based environment.

Suggestion 9-29 — *A bootstrap module can complete the system configuration before entering the main software test control loop.*

The primary purpose of the bootstrap module is to abstract low-level platform specific initialization code, which will only vary by system parameters. The majority of system verification test software can be written independently from parameters such as CPU type and cache configuration, MMU, interrupt controller used, heap and stack size and location and memory controller RAM configurations.

Furthermore, certain debugger applications are able to load code into debug scratch space that can then be executed directly on the target system. This mechanism allows directed tests to be further modularized as self-contained execution elements.

Compilation Process

The guidelines in this section provide recommendations for structuring and compiling the source files that implement the software tests.

Recommendation 9-30 —*A top-level script should manage the full compilation process.*

The compilation process should managed by a top-level Perl script or makefile. A full compilation process will perform the following steps:

1. Visit the entire directory tree under the root directory looking for directories named `testcode`.
2. In all such directories found, locate all C and ASM files in that directory identified by the `.c` and `.s` suffixes, respectively.
3. Compile all C and ASM files to object files in the relevant object code directory if the source is newer than the existing object code file in the object code directory.

Compilation will only be needed if tests have been added or modified and will only recompile the files that were modified since the last compilation. The source files will not need to be recompiled to run specific tests sequences.

Suggestion 9-31 — *The compilation process can support creating a configurable software test image to be loaded into the verification environment.*

It may be desirable to have a common code base that can be reused, compiled and linked to perform a particular function to a particular target, rather than to have many fixed build scripts and fixed test code modules. Tests are added as calls into the `main()` routine. This addition influences the way in which the software image is built, as it requires all elements from the test code library that are used for that particular run. The linker optimizes the test image to contain only those library routines called from `main()`. The software build process should support a mechanism for selecting different test configurations depending on the requirements for a particular run. The test code may be required to test each system peripheral in turn, or the interaction among peripherals, or just concentrate on a single peripheral for any particular test run. A default `main()` configuration would be to include all tests that are defined in the test action library.

Recommendation 9-32 —*The compilation process should maximize portability between target verification environments.*

It may be desirable to support building for different target environments depending on the type of verification to be carried out. Single-peripheral tests may require a high visibility simulation environment for debugging. Full-system or regression test runs may require the runtime performance of an FPGA–based environment. It may be necessary to perform the same tests on different verification environments to ensure that they are conformant with each other.

The test build process must be able to target the same test code to these different environments.

Recommendation 9-33 —*There should be a defined directory structure for the software compilation process.*

A defined directory structure will let the compilation process easily identify and include test code modules. Directory structure is a very subjective topic and it is outside of the scope of this chapter to define a specific structure. The chosen directory structure should be consistent and follow company standards hierarchy and naming conventions.

For illustration purposes, a simple example directory structure is presented below. In this example, files containing test code are placed in a directory named `testcode`. There is one `testcode` directory per peripheral and these directories are located under the root directory.

This example structure shown in Figure 9-6 allows the creation of a *build tree* where the build process traverses the directory tree and locates the test code to be compiled and linked. This structure allows test modules to be associated with their corresponding peripherals and does not restrict the number of peripherals being integrated into the system design.

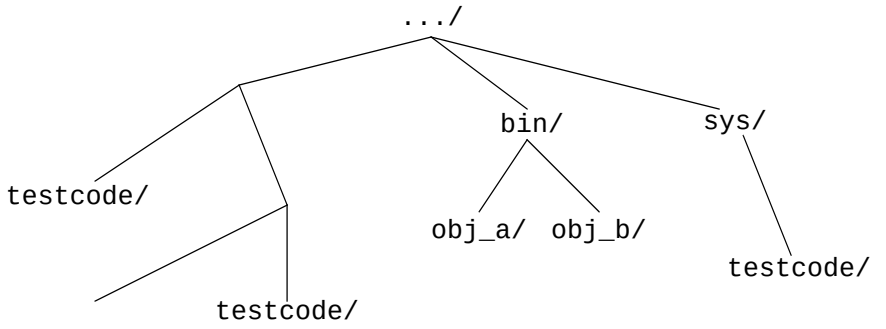


Figure 9-6. Directory Structure for System Software Verification

All scripts supporting the build process would be located in the *bin* directory. The code framework (e.g., bootstrap and support code) is placed in a directory named *testcode* under the *sys* directory.

The use of a separate object files subdirectory in the directory structure from Figure 9-6 is to let objects from different build target configurations exist independently from the source code from which they were built, e.g., simulation vs FPGA build. Separating the object code in this way also allows for creating parallel builds and debug sessions or for keeping known good test runs whilst changing or developing new tests.

Running Tests

The guidelines presented next provide recommendations for identifying which tests to include in a software image and build that image for execution on a target verification environment. After all source files have been compiled, running a test sequence requires:

1. Specifying the tests and test conditions in a top-level source file.
2. Compiling these top-level source files into object files in the relevant object code directory if the source is newer than the existing object code file in the object code directory.
3. Linking all required object files into an appropriate *.elf* or binary files.
4. Running the *.elf* or binary on the chosen target and setting up other binaries as needed.

The guidelines presented in this section describe mechanisms for selecting tests in a directed fashion, selecting test sequences and runtime test selection.

Alternative 9-34 —*Test actions can be invoked directly by using the* *svSYS_ACTION_RUN()* *macro.*

Directed testing can be performed by selecting a specific set of test actions. For example, the following statement will execute the action defined by the *svP456_FooAction* on the P456 peripheral:

```
svSYS_ACTION_RUN(P456, 0, svP456_FooAction)
```

It will be translated into the function call:

```
svSYS_ActionRun("svP456_FooAction", "P456",
&svP456_FooAction,
                svSYS_SystemData[svSYS_Element_P456_0]);
```

Alternative 9-35 —*Test actions can be invoked via action sheets.*

An action sheet identifies all available actions for a given peripheral. The bootstrap module code can then iterate through all tests for a particular peripheral. All tests in an action sheet must target the same peripheral.

An action sheet is specified as a zero-terminated array of *svSYS_ActionSheetItem* elements.

Example 9-8. Zero-Terminated Action Sheet

```
const svSYS_ActionSheetItem ActionSheet_P123 [] = {
   {&MyP123FirstAction,
    &MyP123FirstAction_Level,
    &MyP123FirstAction_Name},
   {&MyP123SecondAction,
    &MyP123SecondAction_Level,
    &MyP123SecondAction_Name},
   {0} //Must be zero terminated
}
```

An action sheet is then executed using the *svSYS_ACTION_SHEET_RUN()* macro. It will iterate through all items of the appropriate complexity level on the action sheet associated with the peripheral in the specified order. For example:

```
svSYS_ACTION_SHEET_RUN(P456, 0, 1, svTestSequence)
```

Suggestion 9-36 — *Tests can be executed in different orders*

It is possible that a certain execution order of test actions can mask problems. The order of execution of all relevant tests in an action sheet is specified by the third argument of the *svSYS_ACTION_SHEET_RUN()* macro. See Appendix D for more details.

Suggestion 9-37 — *Tests from multiple action sheets can be executed.*

A *svSYS_AllActionSheetsRun()* can be implemented to iterate through all peripherals in the system descriptor, in a specified sequence. For each selected peripheral, it executes its associated action sheet.

Suggestion 9-38 — *Tests can be allowed to be chosen and run without recompiling and rebuilding.*

The global variables *svSYS_Peripherals* and *svSYS_Sequence* can be checked by the *svSYS_AllActionSheetsRun()*. If either is zero, the function will return. Otherwise, any time the variables have changed, the function will run itself again, using the values of the global variables as parameters. If the execution of

a test sequence is stopped and new values inserted in the global variables, a new set of tests will be run. Users can thus modify the selection of components and test sequences at run-time using a debugger.

Bootstrap

The bootstrap module performs the following tasks:

1. Do minimal setup.
2. Set up the memory controllers.
3. Do further setup including stacks, heap, caches, MMU (no remapping, using a pre-generated page table).
4. Optionally perform the board configuration.
5. Execute the test(s).
6. Perform end-of-test operations.

After completing the end-of-test operations, the bootstrap module should report a summary of tests. The number of tests passed, skipped and failed should be reported.

The guidelines presented next should be followed when creating a bootstrap module.

Suggestion 9-39 — *The bootstrap module can identify the underlying execution environment.*

If the bootstrap code is common among different target environments, i.e., only needs to change configuration parameters such as timing values, and memory types, the bootstrap module could first ascertain which environment is being targeted and perform the bootstrap operations accordingly.

Any bootstrap functionality that is common to all platforms can be implemented in the bootstrap code module. For example, in a CPU-based design, the bootstrap module will set up caches and memory management units. The bootstrap module should also include any setup that varies only by system parameters, such as stack and heap location and limits.

Recommendation 9-40 —*Configuration parameters should be obtained from a system description address map file.*

Many modern linker applications support the facility to import address map configuration data. The resulting object code can thus be correctly relocated in the system address map. This data should be derived from the system descriptor.

Recommendation 9-41 —*Memory set-up code should not use the stack.*

Because the stack is setup by the same bootstrap module that sets up the memories, it may not have been initialized.

SUMMARY

The importance of the processor integration environment is demonstrated when system components are integrated together as they would be in the final system. Modeling environments aside, this environment is the designer’s first "real" opportunity to trial software, including driver code, on "real" hardware. This environment also helps the system designer reduce risk by exercising software-based tests that qualify predicted bandwidth and latency corner-case scenarios identified in earlier stages of development, e.g., transaction modeling.

The concept of two distinct software test environments, one for component integration testing and the other for system software soak testing has also been introduced, which supports the above goals through a staged approach where confidence in lower-level integration testing can be gained before running more complex system tests.

Using a software test action framework maintains a consistency with the XVC methodology discussed in Chapter 8, and this framework facilitates these more complex system tests, e.g., synchronizing external system stimulus with software states. Using a system descriptor to abstract system resources used by the test software also reduces maintenance and promotes portability to future and derivative designs.

Ultimately, the software test action framework can also be used as a vehicle to carry the verification forward into an end-user FPGA and/or silicon implementations of the design.

APPENDIX A VMM STANDARD LIBRARY SPECIFICATION

This appendix specifies the detailed behavior of a set of base and utility classes that can be used to implement a VMM-compliant verification environment and verification components. The actual implementation of these classes is left to each tool provider. Chapter 4 provides detailed guidelines on how to use these classes.

VMM_ENV

The class is a base class used to implement verification environments. The guidelines covering the implementation of verification environments based on this class can be found in section titled "Simulation Control" on page 124.

`vmm_log log;`

Message service interface for the verification environment. This property is set by the constructor using the specified environment name and may be modified at run time.

```
vmm_notify notify;
enum {GEN_CFG,
      BUILD,
      RESET_DUT,
      CFG_DUT,
      START,
      WAIT_FOR_END,
```

```
    STOP,
    CLEANUP,
```

REPORT};

Notification service interface and predefined notifications used to indicate the progression of the verification environment. The predefined notifications are used to signal the start of the corresponding predefined virtual methods. All notifications are ON/OFF.

`function new(string name = "Verif Env");`

Creates an instance of the verification environment, with the specified name. The name is used as the name of the message service interface.

`task run()`

Runs all remaining steps of the simulation, including *`vmm_env::stop()`*, *`vmm_env::cleanup()`* and *`vmm_env::report()`*. This method must be explicitly invoked in the test programs.

`virtual function void gen_cfg();`

Randomizes the test configuration descriptor. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the *`vmm_env::build()`* method.

`virtual function void build();`

Builds the verification environment according to the value of the test configuration descriptor. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the *`vmm_env::reset_dut()`* method.

`virtual task reset_dut();`

Physically resets the DUT to make it ready for configuration. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the *`vmm_env::cfg_dut()`* method.

`virtual task cfg_dut();`

Configures the DUT according to the value of the test configuration descriptor. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the *`vmm_env::start()`* method.

`virtual task start();`

Starts all the components of the verification environment to start the actual test. If this method has not been explicitly invoked in the test program, it will be implictly invoked by the *`vmm_env::wait_for_end()`* method.

`event end_test;`

Event that, when triggered, should cause the *`vmm_env::wait_for_end()`* method to return. It is up to the user-defined implementation of the *`vmm_env::wait_for_end()`* method to detect that this event has been triggered and return.

`virtual task wait_for_end();`

Waits for an indication that the test has reached completion or its objective—whatever these may be. When this task returns, it signals that the end of simulation condition has been detected. If this method has not been explicitly invoked in the test program, it will be implictly invoked by the *`vmm_env::stop()`* method.

`virtual task stop();`

Stops all the components of the verification environment to terminate the simulation cleanly. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the *`vmm_env::cleanup()`* method.

`virtual task cleanup();`

Performs clean-up operations to let the simulation terminate gracefully. Clean-up operations may include letting the DUT drain off all buffered data, reading statistics registers in the DUT and sweeping the scoreboard for leftover expected responses. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the *`vmm_env::run()`* method.

`virtual task report();`

Reports final success or failure of the test and close all files. If this method has not been explicitly invoked in the test program, it will be implicitly invoked by the *`vmm_env::run()`* method.

VMM_LOG

The *vmm_log* class used implements an interface to the message service. The guidelines covering the usage of the message service can be found in section titled "Message Service" on page 134.

Several methods apply to multiple message service interfaces, not just the one where the method is invoked. All message service interfaces that match the specified name and instance name are affected by these methods. If the name or instance name is enclosed between slashes (e.g., "/.../"), then they are interpreted as *sed*-style regular expressions. If a value of *""* is specified, then the name or instance name of the current message service interface is specified. If the *recurse* parameter is TRUE (i.e., non-zero), then all interfaces logically under the matching message service interfaces are also specified.

```
function new( string name,
              string instance,
              vmm_log under = null);
```

Creates a new instance of a message service interface, with the specified interface name and instance name. Furthermore, a message service interface can optionally be specified as hierarchically below another message service instance to create a logical hierarchy of message service interfaces.

```
virtual function void is_above(vmm_log log);
```

Specifies that this message service instance is hierarchically above the specified message service interface. This method is the corollary of the *under* argument of the constructor and need not be used if the specified message service interface has already been constructed as being under this message service interface.

```
virtual function vmm_log copy(vmm_log to = null);
```

Copies the configuration of this message service interface to the specified message service interface (or a new interface if none is specified) and returns a reference to the interface copy. The current configuration of the message service interface is copied, except the hierarchical relationship information, which is not modified.

```
virtual function string get_name();
virtual function string get_instance();
```

Returns the name and instance name of the message service interface.

```
virtual function void list( string name = "/./",
                            string instance = "/./",
                            bit recurse = 0);
```

Lists all message service interfaces that match the specified name and instance name. If the *recurse* parameter is TRUE (i.e., non-zero), then all interfaces logically under the matching message service interface are also listed.

```
enum { FAILURE_TYP,
       NOTE_TYP,
       DEBUG_TYP,
       TIMING_TYP,
       XHANDLING_TYP,
       REPORT_TYP,
       PROTOCOL_TYP,
       TRANSACTION_TYP,
       COMMAND_TYP,
       CYCLE_TYP,
       INTERNAL_TYP,
       DEFAULT_TYP,
       ALL_TYPS};
```

Enumerated type defining symbolic values for message types used when specifying a message type in properties or method arguments. Table 4-1 on page 135 describes the purpose of the various message types. The *vmm_log::DEFAULT_TYP* and *vmm_log::ALL_TYPS* are special symbolic values usable only with some control methods and are not used to issue actual messages. Multiple message types can be specified to some control methods by combining the value of the required types using the bitwise-or or addition operator.

```
enum { FATAL_SEV,
       ERROR_SEV,
       WARNING_SEV,
       NORMAL_SEV,
       TRACE_SEV,
       DEBUG_SEV,
       VERBOSE_SEV,
       DEFAULT_SEV,
       ALL_SEVS};
```

Enumerated type defining symbolic values for message severities used when specifying a message severity in properties or method arguments. Table 4-2 on page 136 describes the purpose of the various message severities. The *vmm_log::DEFAULT_SEV* and *vmm_log::ALL_SEVS* are special symbolic values usable only with some control methods and are not used to issue actual

messages. Multiple message severities can be specified to some control methods by combining the value of the required severities using the bitwise-or or addition operator.

```
enum { IGNORE,
       CONTINUE,
       DUMP_STACK,
       STOP_PROMPT,
       DEBUGGER,
       COUNT_ERROR,
       ABORT_SIM,
       DEFAULT_HANDLING};
```

Enumerated type defining symbolic values for simulation handling used when specifying a new simulation handling when promoting or demoting a message using the *vmm_log::modify()* method.

Unless otherwise specified, message types are assigned the following default severity and simulation handling:

Table A-1. Default Message Severities and Handling

Message Type	Default Severity	Default Handling
FAILURE_TYP	ERROR_SEV	COUNT_ERROR
NOTE_TYP	NORMAL_SEV	CONTINUE
DEBUG_TYP	DEBUG_SEV	CONTINUE
TIMING_TYP *XHANDLING_TYP*	WARNING_SEV	CONTINUE
TRANSACTION_TYP *COMMAND_TYP*	TRACE_SEV	CONTINUE
REPORT_TYP *PROTOCOL_TYP*	DEBUG_SEV	CONTINUE
CYCLE_TYP	VERBOSE_SEV	CONTINUE
Any type	FATAL_SEV	ABORT_SIM

virtual function vmm_log_format
set_format(vmm_log_format fmt);

Globally sets the message formatter to the specified message formatter instance. A reference to the previously used message formatter instance is returned. A default global message formatter is provided.

virtual function string set_typ_image(int typ,
string image);

Globally replaces the image used to display the specified message type with the specified image. The previous image is returned. Default images are provided.

virtual function string set_sev_image(int sev,
string image);

Globally replaces the image used to display the specified message severity with the specified image. The previous image is returned. Default images are provided.

Example A-1. Colorizing the severity display on ANSI terminals

```
log.set_sev_image(vmm_log::WARNING,
                  "\033[33mWARNING\033[0m");
log.set_sev_image(vmm_log::ERROR_SEV,
                  "\033[31mERROR\033[0m");
log.set_sev_image(vmm_log::FATAL_SEV,
                   "\033[41m*FATAL*\033[0m");
```

virtual function bit start_msg(int typ,
int sev = DEFAULT_SEV);

Prepares to issue a message of the specified type and severity. If the message service interface is configured to ignore messages of the specified type or severity, the function returns FALSE. It returns TRUE otherwise.

virtual function bit text(string msg = “”);

Adds the specified text to the message being constructed. This method specifies a single line of message text and a newline character is automatically appended when the message issued. Additional lines of messages can be produced by calling this method multiple times, once per line. Each additional line is prefixed with the prefix specified in the *vmm_log::format()* method. If an empty string is specified as message text, all previously specified lines of text are flushed to the output, but the message is not terminated. This method may return FALSE if the message will be filtered out based on the text.

A message must be flushed and terminated by calling the *vmm_log::end_msg()* method to trigger the message display and the simulation handling. A message can be flushed multiple times by calling the *vmm_log::text("")* method, but the simulation handling and notification will take effect on the message termination.

If additional lines are produced using the *$display()* system task or other display mechanisms, they will not be considered by the filters, nor included in explicit log files. They may also be displayed out of order if they are produced before the previous lines of the message are flushed.

For single-line messages, the *‘vmm_fatal()*, *‘vmm_error()*, *vmm_warning()*, *‘vmm_note()*, *‘vmm_trace()*, *‘vmm_debug()*, *‘vmm_verbose()*, *‘vmm_report()*, *‘vmm_command()*, *‘vmm_transaction()*, *‘vmm_protocol()* and *‘vmm_cycle()* macros can be used as a shorthand notation.

Table A-2. Message Type and Severity for Shorthand Macros

Macro	Message Type	Message Severity
‘vmm_fatal(vmm_log log, string txt);	Failure	Fatal
‘vmm_error(vmm_log log, string txt);	Failure	Error
‘vmm_warning(vmm_log log, string txt);	Failure	Warning
‘vmm_note(vmm_log log, string txt);	Note	Default
‘vmm_trace(vmm_log log, string txt);	Debug	Trace
‘vmm_debug(vmm_log log, string txt);	Debug	Debug
‘vmm_verbose(vmm_log log, string txt);	Debug	Verbose
‘vmm_report(vmm_log log, string txt);	Report	Default
‘vmm_command(vmm_log log, string txt);	Command	Default

Table A-2. Cont.

Macro	Message Type	Message Severity
‘vmm_transaction(vmm_log log, string txt);	Transaction	Default
‘vmm_protocol(vmm_log log, string txt);	Protocol	Default
‘vmm_cycle(vmm_log log, string txt);	Cycle	Default

virtual function void end_msg();

Flushes and terminates the current message and triggers the message display and the simulation handling. A message can be flushed multiple times using the *vmm_log::text("")* method, but the simulation handling and notification will only take effect on message termination.

```
virtual function void enable_types(  int typs,
                                     string name = "",
                                     string inst = "",
                                     bit recursive = 0);

virtual function void disable_types( int typs,
                                     string name = "",
                                     string inst = "",
                                     bit recursive = 0);
```

Specifies the message types to be displayed/disabled by the specified message service interfaces. Message service interfaces are specified by value or regular expression for both the name and instance name. If no name and no instance are explicitly specified, this message service interface is implicitly specified.

If the name or instance named are specified between “/” characters, then the specification is interpreted as a regular expression that must be matched against all known names and instance names, respectively. Both names must match to consider a message service interface as specified. If the *recursive* argument is TRUE, all message service interface hierarchically below the specified message service interfaces are included in the specification, whether their name and instance name matches or not. A message service interface must exist to be specified.

The *types* argument specifies the messages types to enable or disable. Types are specified as the bitwise-or or sum of all relevant types.

By default, all message types are enabled.

```
virtual function void set_verbosity(int severity,
                                    string name = "",
                                    string inst = "",
                                    bit recursive = 0);
```

Specify the minimum message severity to be displayed when sourced by the specified message service interfaces. See the documentation for the *vmm_enable_types()* method for the interpretation of the *name*, *inst* and *recursive* argument and how they are to specify message service interfaces.

The default minimum severity can be changed by using the "*+vmm_log_default=<sev>*" runtime command-line option, where "*<sev>*" is the desired minimum severity and is a one of the following: "*error*," "*warning*," "*normal*," "*trace*," "*debug*" or "*verbose*". The default verbosity level can be later modified using this method.

The minimum severity level can be globally forced by using the "*+vmm_force_verbosity=<sev>*" runtime command-line option. The specified verbosity overrides the verbosity level specified using this method.

```
virtual function int get_verbosity();
```

Returns the minimum message severity to be displayed when sourced by this message service interface.

```
virtual function int
   modify( string name = "",
           string inst = "",
           bit recursive = 0,
           int typs = ALL_TYPS,
           int severity = ALL_SEVS,
           string text = "/./",
           int new_typ = UNCHANGED,
           int new_severity = UNCHANGED,
           int handling = UNCHANGED);
```

Modifies the specified message source by any of the specified message service interfaces with the new specified type, severity or simulation handling. The message

can be specified by type, severity, numeric ID or by text pattern. By default, messages of any type, severity, ID or text is specified. A message must match all specified criteria.

This method returns a unique message modifier identifier that can be used to remove it using the *vmm_log::unmodify()* method. All message modifiers are applied in the same order they were defined before a message is issued.

```
virtual function void unmodify( int mod_id = -1,
                                string name = "",
                                string instance = "",
                                bit recursive = 0);
```

Removes the specified message modification from the specified message service interfaces. By default, all message modifications are removed.

```
virtual function void log_start( int file,
                                 string name = "",
                                 string instance = "",
                                 bit recurse = 0)
```

Appends all messages produced by the specified message service interfaces to the specified file. The *file* argument must be a file descriptor, as returned by the *$fopen()* system task. By default, all message service interfaces append their messages to the standard output. Specifying a new output file does not stop messages from being appended to previously specified files.

```
virtual function void log_stop( int file,
                                string name = "",
                                string instance = "",
                                bit recurse = 0);
```

Messages issued by the specified message service interfaces are no longer appended to the specified file. The *file* argument must be a file descriptor, as returned by the $*fopen*() system task. If the specified *file* argument is *0*, messages are no longer sent to the standard simulation output and transcript. If the *file* argument is specified as –1, appending to all files, except the standard output, is stopped.

```
virtual function void stop_after_n_errors(int n);
```

Aborts the simulation after the specified number of messages with a simulation handling of *COUNT_ERROR* has been issued. This value is global and all messages from any message service interface count toward this limit. A zero or negative value

specifies no maximum. The default value is 10. The message specified by the *vmm_log_format::abort_on_error()* is displayed before the simulation is aborted.

```
virtual function int
          get_message_count( int severity = ALL_SEVS,
                             string name = "",
                             string instance = "",
                             bit recurse = 0);
```

Returns the total number of messages of the specified severities that have been issued from the specified message service interfaces. Message severities can be specified as a sum of individual message severities to specify more than one severity.

```
virtual function int
          create_watchpoint(int types = ALL_TYPS,
                            int severity = ALL_SEVS,
                            string text = "",
                            logic issued = 1'bx);
```

Creates a watchpoint descriptor that will be triggered when the specified message is used. The message can be specified by type, severity or by text pattern. By default, messages of all types, severities and text are specified. A message must match all specified criteria to trigger the watchpoint. The *issued* parameter specifies if the watchpoint is triggered when the message is physically issued (1'b1), physically not issued, i.e., filtered out (1'b0) or regardless if the message is physically issued or not (1'bx).

A watchpoint will be repeatedly triggered every time a message matching the watchpoint specification is issued by a message service interface associated with the watchpoint.

```
virtual function void
           add_watchpoint(   int watchpoint_id,
                             string name = "",
                             string instance = "",
                             bit recurse = 0);
virtual function void
           remove_watchpoint(int watchpoint_id,
                             string name = "",
                             string instance = "",
                             bit recurse = 0);
```

Adds or removes the specified watchpoint to or from the specified message service interfaces. If a message matching the watchpoint specification is issued by one of the specified message service interfaces associated with the watchpoint, the watchpoint will be triggered.

```
virtual task wait_for_watchpoint( int watchpoint_id,
                                  ref vmm_log_msg msg);
```

Waits for the specified watchpoint to be triggered by a message issued by one of the message service interfaces attached to the watchpoint. A descriptor of the message that triggered the watchpoint will be returned.

```
virtual task wait_for_msg( string name = "",
                           string instance = "",
                           bit recurse = 0,
                           int typs = ALL_TYPS,
                           int severity = ALL_SEVS,
                           string text = "",
                           logic issued = 1'bx,
                           ref vmm_log_msg msg);
```

Sets up and waits for a one-time watchpoint for the specified message on the specified message service interface. The watchpoint is triggered only once and removed after being triggered.

```
virtual task report( string name = "/./",
                     string instance = "/./",
                     bit recurse = 0);
```

Reports a failure if any of the specified message service interfaces have issued any error or fatal messages. Reports a success otherwise. The text of the pass or fail message is specified using the *`vmm_log_format::pass_or_fail()`* method.

```
virtual function void
          prepend_callback(vmm_log_callbacks cb);
virtual function void
          append_callback(vmm_log_callbacks cb);
```

Globally prepends or appends the specified callback façade instance with the message service. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once. Callback façade instances can be unregistered and re-registered dynamically.

```
virtual function void
          unregister_callback(vmm_log_callbacks cb);
```

Globally unregisters the specified callback façade instance with the message service. A warning is issued if the specified façade instance is not currently registered with the service. Callback façade instances can later be re-registered.

vmm_log_msg

This class describes a message issued by a message service interface that caused a watchpoint to be triggered. It is returned by the *vmm_log::wait_for_watchpoint()* and *vmm_log::wait_for_msg()* method.

```
vmm_log log;
```

A reference to the message reporting interface that has issued the message.

```
time timestamp;
```

The simulation time when the message was issued.

```
int original_typ;
```

Original message type as specified in the code creating the message.

```
int original_severity;
```

Original message severity as specified in the code creating the message.

```
int effective_typ;
```

Effective message type as potentially modified by the *vmm_log::modify()* method.

`int effective_severity;`

Effective message severity as potentially modified by the *`vmm_log::modify()`* method.

`string text[$];`

Formatted text of the message. Each element of the array contains one line of text as built by individual calls to the *`vmm_log::text()`* method.

`bit issued;`

Indicates if the message has been physically issued or not. If non-zero, then the message has been issued.

`int handling;`

The simulation handling after the message is physically issued.

vmm_log_format

This class is used to specify how messages are formatted before being displayed or logged to files. The default implementation of these methods produces the default message format.

```
virtual function string format_msg( string name,
                                    string instance,
                                    string msg_typ,
                                    string severity,
                                    string lines[$]);
```

This method is called by all message service interfaces to format a message on the first occurrence of a call to the *`vmm_log::end_msg()`* method or empty *`vmm_log::text("")`* method call. Subsequent calls to the *`vmm_log::end_msg()`* method or empty *`vmm_log::text("")`* method use the *`vmm_log_format::continue_msg()`* method.

The *`lines`* parameter contains one line of message text for each non-empty call to the *`vmm_log::text()`* method.

```
virtual function string continue_msg( string name,
                                      string instance,
                                      string msg_typ,
                                      string severity,
                                      string lines[$]);
```

This method is called by all message service interfaces to format a message on the first occurrence of a call to the *vmm_log::end_msg()* method or empty *vmm_log::text("")* method call. Subsequent calls to the *vmm_log::end_msg()* method or empty *vmm_log::text("")* method use the *vmm_log_format::continue_msg()* method.

The *lines* parameter contains one line of message text for each non-empty call to the *vmm_log::text()* method since the last empty call to the *vmm_log::text("")* method. It does not contain lines that were previously formatted in a prior call to this method or the *vmm_log_format::format_msg()* method.

```
virtual function string abort_on_error( int count,
                                        int limit);
```

This method is called when the total number of *COUNT_ERROR* messages exceed the error message threshold. The string returned by the method describes the cause of the simulation aborting. If *null* is returned, no explanation is displayed.

This method is called and the returned string is displayed before the *vmm_log_callbacks::pre_abort()* callback methods are invoked.

```
virtual function string pass_or_fail( bit pass,
                                      string name,
                                      string instance,
                                      int fatals,
                                      int errors,
                                      int warnings,
                                      int dem_errs,
                                      int dem_warns);
```

This method is called by the *vmm_log::report()* method to format the final pass/fail message at the end of simulation. The *pass* argument, if true, indicates that the simulation was successful. The *name* and *instance* arguments are the specified name and instance names specified to the *vmm_log::report()* method. The *fatals* argument is the total number of *vmm_log::FATAL_SEV* messages that were issued. The *errors* argument is the total number of *vmm_log::ERROR_SEV* messages that were issued. The *warnings* argument is

the total number of *vmm_log::WARNING_SEV* messages that were issued. The *dem_errs* argument is the total number of *vmm_log::ERROR_SEV* messages that were demoted. The *dem_warns* argument is the total number of *vmm_log::WARNING_SEV* messages that were demoted.

vmm_log_callbacks

This class provides a façade for the callback methods provided by the message service. Callbacks are associated with the message service itself, not a particular message service interface instance.

virtual task pre_abort(vmm_log log);

This callback method is invoked by the message service before the simulation is aborted because of an *ABORT* simulation handling or exceeded maximum number of *COUNT_ERROR* messages. The message service instance provided as argument can be used to issue further messages.

virtual task pre_stop(vmm_log log);

This callback method is invoked by the message service before the simulation is stopped because of a *STOP* simulation handling. The message service instance provided as argument can be used to issue further messages.

virtual task pre_debug(vmm_log log);

This callback method is invoked by the message service before the breaking into the debugger because of a *DEBUGGER* simulation handling. The message service instance provided as argument can be used to issue further messages.

Example A-2. Issuing a Simple Message

```
program test;
   vmm_log log = new("Test", "Singleton");

initial begin
   if (log.start_msg(vmm_log::DEBUG_TYP)) begin
      void'(log.text("Starting test"));
      log.end_msg();
   end
   ...
end
endprogram
```

Example A-3. Issuing a Simple Message using a Macro

```
program test;
   vmm_log log = new("Test", "Singleton");

initial begin
   ‘vmm_debug(log, "Starting test");
   ...
end
endprogram
```

Example A-4. Issuing a Complex Message

```
program test;
   vmm_log log = new("Test", "Singleton");

initial begin
   ...
   while (log.start_msg(vmm_log::FAILURE_TYP,
                        vmm_log::WARNING_SEV)) begin
       string str;
       if (!log.text(...)) break;
       if (!log.text(transaction.psdisplay())) break;
       $sformat(str, ...);
       if (!log.text(str)) break;
       log.end_msg();
       break;
   end
   ...
end
endprogram
```

Example A-5. Pattern-Based Message Promotion

Demote all messages with an *ERROR_SEV* severity containing the pattern “abort” in all instances of message service interfaces named “AMBA AHB Interface Master” to a *WARNING_SEV* severity.

```
program test
   verif_env env = new;

initial begin
   env.build();
   env.log.modify("AMBA AHB Interface Master", "/./", ,
                  , , vmm_log::ERROR_SEV, "/abort/",
                  , vmm_log::WARNING_SEV);
   ...
end
endprogram
```

VMM_DATA

This base class is to be used as the basis for all transaction descriptors and data models. It provides a standard set of methods expected to be found in all descriptors. It also creates a common class—akin to C's *void* type—that can be used to create generic components. The guidelines covering the development of data and transaction descriptors based on this class can be found in section titled "Data and Transactions" on page 140.

function new(vmm_log log);

Creates a new instance of this data model or transaction descriptor with the specified message service interface. The specified message service interface is used when constructing the *vmm_data::notify* property.

Because of the potentially large number of instances of data objects, a *class-static* message service interface should be used to minimize memory usage and to be able to control class-generic messages:

```
class eth_frame extends vmm_data {
   static vmm_log log = new("eth_frame", "class");
   function new()
      super.new(this.log);
      ...
   endfunction
endclass: eth_frame
```

function vmm_log set_log(vmm_log log);

Replaces the message service interface for this instance of a data model or transaction descriptor with the specified message service interface and returns a reference to the previous message service interface. Can be used to associate a descriptor with the message service interface of a transactor currently processing the transaction or to set the service when it was not available during initial construction.

int stream_id;
int scenario_id;
int data_id;

Unique identifiers for a data model or transaction descriptor instance. They specify the offset of the descriptor within a sequence and the sequence offset within a stream. These properties must be set by the transactor that instantiates the descriptor. They are set by the predefined generator before randomization so they can be used to specify conditional constraints to express instance-specific or stream-specific constraints.

```
vmm_notify notify;
enum { EXECUTE;
       STARTED;
       ENDED};
```

A notification service interface with three pre-configured events. The *EXECUTE* notification is ON/OFF and indicated by default. It can be used to prevent the execution of a transaction or the transfer of data if reset. The *STARTED* and *ENDED* notifications are ON/OFF events and indicated by the transactor at the start and end of the transaction execution or data transfer. The meaning and timing of the notifications is specific to the transactor executing the transaction described by this instance.

```
function void display(string prefix = "");
```

Displays the current value of the transaction or data described by this instance in a human-readable format on the standard output. Each line of the output will be prefixed with the specified prefix. This method prints the value returned by the *psdisplay()* method.

```
virtual function string psdisplay(string prefix = "");
```

Returns an image of the current value of the transaction or data described by this instance in a human-readable format as a string. The string may contain newline characters to split the image across multiple lines. Each line of the output must be prefixed with the specified prefix.

```
virtual function bit is_valid(bit silent = 1,
                              int kind = -1);
```

Checks if the current value of the transaction or data described by this instance is valid and error-free, according to the optionally specified kind or format. Returns TRUE (i.e., non-zero) if the content of the object is valid. Returns FALSE otherwise. The meaning (and use) of the *kind* argument is descriptor-specific and defined by the user-extension of this method.

If *silent* is TRUE (i.e., non-zero), no error or warning messages are issued if the content is invalid. If silent is FALSE, warning or error messages may be issued if the content is invalid.

```
virtual function vmm_data allocate();
```

Allocates a new instance of the same type as the object instance. Returns a reference to the new instance. Useful to implement class factories to create instances of user-defined derived class in generic code written using the base class type.

`virtual function vmm_data copy(vmm_data to = null);`

Copies the current value of the object instance to the specified object instance. If no target object instance is specified, a new instance is allocated. Returns a reference to the target instance.

Note that the following trivial implementation will not work. Constructor copying is a shallow copy. The objects instantiated in the object (such as those referenced by the *log* and *notify* properties) are not copied and both copies will share references to the same service interfaces. Furthermore, it will not properly handle the case when the to argument is not null.

Example A-6. Invalid Implementation of the *vmm_data::copy()* Method

```
function vmm_data atm_cell::copy(vmm_data to = null)
   copy = new this;
endfunction
```

The following implementation is usually preferable:

Example A-7. Proper Implementation of the *vmm_data::copy()* Method

```
function vmm_data atm_cell::copy(vmm_data to = null)
   atm_cell cpy;

   if (to != null) begin
      if ($cast_assign(cpy, to)) begin
         ‘vmm_fatal(log, “Not a atm_cell instance”);
         return;
      end
   end else cpy = new;

   this.copy_data(cpy);
   cpy.vpi = this.vpi;
   ...
   copy = cpy;
endfunction: copy
```

The base-class implementation of this method must not be called as it contains error detection code of a derived class that forgot to supply an implementation. The *vmm_data::copy_data()* method should be called instead.

`virtual protected function void copy_data(vmm_data to);`

Copies the current value of all base-class data properties in the current data object into the specified data object instance. This method should be called by the

implementation of the *vmm_data::copy()* method in classes immediately derived from this base class.

```
virtual function bit compare( input vmm_data to,
                              output string diff,
                              input int kind = -1);
```

Compares the current value of the object instance with the current value of the specified object instance, according to the specified kind. Returns TRUE (i.e., non-zero) if the value is identical. If the value is different, FALSE is returned and a descriptive text of the first difference found is returned in the specified *string* variable. The *kind* argument may be used to implement different comparison functions (e.g., full compare, comparison of *rand* properties only, comparison of all properties physically implemented in a protocol and so on.)

```
virtual function int unsigned byte_pack(
                                ref logic [7:0] bytes[],
                                int unsigned offset = 0,
                                int kind = -1);
```

Packs the content of the transaction or data into the specified dynamic array of bytes, starting at the specified offset in the array. The array is resized appropriately. Returns the number of bytes added to the array.

If the data can be interpreted or packed in different ways, the *kind* argument can be used to specify which interpretation or packing to use.

```
virtual function int unsigned byte_unpack(
                            const ref logic [7:0] bytes[],
                            input int unsigned offset = 0,
                            input int len = -1,
                            input int kind = -1);
```

Unpacks the specified number of bytes of data from the specified offset in the specified dynamic array into this descriptor. If the number of bytes to unpack is specified as *-1*, the maximum number of bytes will be unpacked. Returns the number of bytes unpacked. If there is not enough data in the dynamic array to completely fill the descriptor, the remaining properties are set to unknown and a warning may be issued.

If the data can be interpreted or unpacked in different ways, the *kind* argument can be used to specify which interpretation or packing to use.

`virtual function int unsigned byte_size(int kind = -1);`

Returns the number of bytes required to pack the content of this descriptor. This method will be more efficient than *`vmm_data::byte_pack()`* for simply knowing how many bytes are required by the descriptor because no packing is actually done.

If the data can be interpreted or packed in different ways, the *`kind`* argument can be used to specify which interpretation or packing to use.

`virtual function int unsigned max_byte_size(`
`int kind = -1);`

Returns the maximum number of bytes that will ever be required to pack the content of any instance of this descriptor. A value of 0 indicates an unknown maximum size. Can be used to allocate memory buffers in the DUT or verification environment of suitable sizes.

If the data can be interpreted or packed in different ways, the *`kind`* argument can be used to specify which interpretation or packing to use.

`virtual function void save(int file);`

Appends the content of this descriptor to the specified file. The format is user defined and may be binary. By default, simply packs the descriptor and saves the value of the bytes, in sequence, as binary values and terminated by a newline.

`virtual function bit load(int file);`

Sets the content of this descriptor from the data in the specified file. The format is user defined and may be binary. By default, interprets a complete line as binary byte data and unpacks it.

Should return FALSE (i.e., zero) if the loading operation was not successful.

VMM_CHANNEL

This class implements a generic transaction-level interface mechanism. The guidelines covering the usage of the channel can be found in section titled "Transaction-Level Interfaces" on page 171.

Offset values, either accepted as arguments or returned values, are always interpreted the same way. A value of 0 indicates the head of the channel (first transaction descriptor added). A value of –1 indicates the tail of the channel (last transaction descriptor added). Positive offsets are interpreted from the head of the channel. Negative offsets are interpreted from the tail of the channel. For example, an offset value of –2 indicates the transaction descriptor just before the last transaction descriptor in the channel. It is illegal to specify a non-zero offset that does not correspond to a transaction descriptor already in the channel.

The channel includes an active slot that can be used to create more complex transactor interfaces. The active slot counts toward the number of transaction descriptors currently in the channel for control-flow purposes but cannot be accessed nor specified via an offset specification.

The implementation uses a macro to define a class named "<*class_name*>*_chan*" derived from the class named "*vmm_channel*" for any user-specified class named "*class_name*".

`vmm_channel(class_name);`

Defines a channel class to transport instances of the specified class. The transported class must be derived from the *vmm_data* class. This macro is typically invoked in the same file where the specified class is defined and implemented.

This macro creates an external class declaration and no implementation. It is typically invoked when the channel class must be visible to the compiler but the actual channel class declaration is not yet available.

```
function new( string name,
              string instance,
              int unsigned full = 1,
              int unsigned empty = 0,
              bit fill_as_bytes = 0);
```

Creates a new instance of a channel with the specified name, instance name and full and empty levels. If the *fill_as_bytes* argument is TRUE (i.e., non-zero) the full and empty levels and the fill level of the channel are interpreted as the number of bytes in the channel as computed by the sum of *vmm_data::byte_size()* of all transaction descriptors in the channel, not the number of objects in the channel. If the value is FALSE (i.e., zero), the full and empty levels and the fill level of the channel are interpreted as the number of transaction descriptors in the channel. It is illegal to configure a channel with a full level lower than the empty level.

`vmm_log log;`

Message service interface for messages issued from within the channel instance.

```
function void reconfigure( int full = -1,
                           int empty = -1,
                           logic fill_as_bytes = 1'bx);
```

If not negative, reconfigure the full or empty levels of the channel to the specified levels. Reconfiguration may cause threads currently blocked on a *`vmm_channel::put()`* call to unblock. If the *`fill_as_bytes`* argument is specified as *`1'b1`* or *`1'b0`*, the interpretation of the fill level of the channel is modified accordingly. Any other value leaves the interpretation of the fill level unchanged.

```
function int unsigned full_level();
function int unsigned empty_level();
```

Returns the currently configured full or empty level.

```
function int unsigned level();
```

Returns the current fill level of the channel. The interpretation of the fill level depends on the configuration of the channel instance.

```
function int unsigned size();
```

Returns the number of transaction descriptors currently in the channel, including the active slot, regardless of the interpretation of the fill level.

```
function bit is_full();
```

Returns TRUE (i.e., non-zero) if the fill level is greater than or equal to the currently configured full level. Returns FALSE otherwise.

`vmm_notify notify`

An event notification interface used to indicate the occurrence of significant events within the channel. The notifications shown in Table A-3 are pre-configured.

Table A-3. Pre-Configured Notifications in *vmm_channel* Notifier Interface

Symbolic Property	Corresponding Significant Event
vmm_channel::FULL	Channel has reached or surpassed its configured full level. This notification is configured ON/OFF. No status is returned.
vmm_channel::EMPTY	Channel has reached or underflowed the configured empty level. This event is configured ON/OFF. No status is returned.
vmm_channel::PUT	A new transaction descriptor has been added to the channel. This event is configured ONE_SHOT. The newly added transaction descriptor is available as status.
vmm_channel::GOT	A transaction descriptor has been removed from the channel. This event is configured ONE_SHOT. The newly removed transaction descriptor is available as status.
vmm_channel::PEEKED	A transaction descriptor has been peeked from the channel. This event is configured ONE_SHOT. The newly peeked transaction descriptor is available as status.
vmm_channel::ACTIVATED	A transaction descriptor has been transferred to the active slot. This notification also implies a *PEEKED* notification. This event is configured ONE_SHOT. The newly activated transaction descriptor is available as status.
vmm_channel::ACT_STARTED	The state of a transaction descriptor in the active slot has been updated to *STARTED*. This event is triggered ONE_SHOT. The currently active transaction descriptor is available as status.
vmm_channel::ACT_COMPLETED	The state of a transaction descriptor in the active slot has been updated to *COMPLETED*. This event is configured ONE_SHOT. The currently active transaction descriptor is available as status.

Table A-3. Cont.

Symbolic Property	Corresponding Significant Event
vmm_channel::ACT_REMOVED	A transaction descriptor has been removed from the active slot. This notification also implies a *GOT* notification. This event is configured ONE_SHOT. The newly removed transaction descriptor is available as status.
vmm_channel::LOCKED	A side of the channel has been locked. This event is configured ONE_SHOT.
vmm_channel::UNLOCKED	A side of the channel has been unlocked. This event is configured ONE_SHOT.

function void flush();

Flushes the content of the channel. Flushing will unblock any thread currently blocked in the *vmm_channel::put()* method. This method will cause the *FULL* notification to be reset or the *EMPTY* notification to be indicated. Flushing a channel unlocks all sources and consumers.

function void sink();

Flushes the content of the channel and sinks any further objects put into it. No transaction descriptors will accumulate in the channel while it is sunk. Any thread attempting to obtain a transaction descriptor from the channel will be blocked until the flow through the channel is restored using the *vmm_channel::flow()* method. This method will cause the *FULL* notification to be reset or the *EMPTY* notification to be indicated.

function void flow();

Restores the normal flow of transaction descriptors through the channel.

function void lock(bit [1:0] who);
function void unlock(bit [1:0] who);

Blocks any source (consumer) as if the channel was full (empty) until explicitly unlocked. The side that is to be locked or unlocked is specified using the sum of the symbolic values shown in Table A-4.

Table A-4. Channel Endpoint Identifiers

Symbolic Property	Channel Endpoint
vmm_channel::SOURCE	The producer side, i.e., any thread calling the *vmm_channel::put()* method
vmm_channel::SINK	The consumer side, i.e., any thread calling the *vmm_channel::get()* method

Locking a source does not indicate the *FULL* notification, nor does locking the sink indicate the *EMPTY* notification—although they have the same control-flow effect.

function bit is_locked(bit [1:0] who);

Returns TRUE (i.e., non-zero) if any of the specified sides is locked. If both sides are specified, returns TRUE if any side is locked.

Example A-8. Querying the Lock Status of a Channel

```
while (chan.is_locked(vmm_channel::SOURCE +
                      vmm_channel::SINK)) begin
   chan.notify.wait_for(vmm_channel::UNLOCKED);
end
```

task put(*class_name* obj,
int offset = -1);

Puts the specified transaction descriptor in the channel at the specified offset. If the fill level of the channel, including the active slot, is greater than or equal to the configured full level, or if the source is locked, the task will block until the fill level of the channel is less than or equal to the configured empty level and the source is unlocked.

It is an error to specify an offset that does not already exist in the channel.

This method may cause the *FULL* notification to be indicated and will cause the *EMPTY* notification to be reset.

function void sneak(*class_name* obj,
int offset = -1)

Puts the specified transaction descriptor in the channel at the specified offset. This task will never block, regardless of the configured full level. Use only when a guaranteed non-blocking version of *vmm_channel::put()* is required—for

example, inside a *function*—and threads using this method have some other means of eventually blocking their execution.

It is an error to specify an offset that does not already exist in the channel or sneak a new transaction descriptor into a locked channel.

This method may cause the *FULL* notification to be indicated and will cause the *EMPTY* notification to be reset.

function *class_name* unput(int offset = -1);

Removes the specified transaction descriptor from the channel. It is an error to specify an offset to a transaction descriptor that does not exist.

This method may cause the *EMPTY* notification to be indicated and will cause the *FULL* notification to be reset.

```
task get( output class_name obj,
          input int offset = 0);
```

Retrieves the next transaction descriptor in the channel at the specified offset. If the channel is empty, the function will block until a transaction descriptor is available to be retrieved. This method may cause the *EMPTY* notification to be indicated or the *FULL* notification to be reset.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or with a non-empty active slot.

```
task peek( output class_name obj,
           input int offset = 0);
```

Gets a reference to the next transaction descriptor that will be retrieved from the channel at the specified offset without actually retrieving it. If the channel is empty, the function will block until a transaction descriptor is available to be retrieved.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or with a non-empty active slot.

```
task activate( output class_name obj,
               input int offset = 0);
```

If the active slot is not empty, first removes the transaction descriptor currently in the active slot.

Move the transaction descriptor at the specified offset in the channel to the active slot and update the status of the active slot to *vmm_channel::PENDING*. If the channel is empty, this method will wait until a transaction descriptor becomes available. The transaction descriptor is still considered as being in the channel.

It is an error to invoke this method with an offset value greater than the number of transaction descriptors currently in the channel or to use this method with multiple concurrent consumer threads.

function *class_name* active_slot();

Returns the transaction descriptor currently in the active slot. Returns *null* if the active slot is empty.

function *class_name* start();

Updates the status of the active slot to *vmm_channel::STARTED*. The transaction descriptor remains in the active slot. It is an error to call this method if the active slot is empty. The *vmm_data::STARTED* notification of the transaction descriptor in the active slot is indicated.

function *class_name* complete(vmm_data status = null);

Updates the status of the active slot to *vmm_channel::COMPLETED*. The transaction descriptor remains in the active slot and may be restarted. It is an error to call this method if the active slot is empty. The *vmm_data::ENDED* notification of the transaction descriptor in the active slot is indicated with the optionally specified completion status descriptor.

function *class_name* remove();

Updates the status of the active slot to *vmm_channel::INACTIVE* and removes the transaction descriptor from the active slot from the channel. This method may cause the *EMPTY* notification to be indicated or the *FULL* notification to be reset. It an error to call this method with an active slot in the *vmm_channel::STARTED* state. The *vmm_data::ENDED* notification of the transaction descriptor in the active slot is indicated.

function active_status_e status();

Returns one of the enumerated values in Table A-5, indicating the status of the transaction descriptor in the active slot.

Table A-5. Pre-Configured Notifications in *vmm_channel* Notifier Interface

Symbolic Property	Corresponding Significant Event
vmm_channel::INACTIVE	No transaction descriptor is present in the active slot.
vmm_channel::PENDING	A transaction descriptor is present in the active slot but it has not been started yet.
vmm_channel::STARTED	A transaction descriptor is present in the active slot and it has been started, but it is not completed yet. The transaction is being processed by the downstream transactor
vmm_channel::COMPLETED	A transaction descriptor is present in the active slot and it has been processed by the downstream transactor, but it has not yet been removed from the active slot.

`task tee(output class_name obj);`

When the tee mode is ON, retrieve a copy of the transaction descriptor references that have been retrieved by the *get()* or *activate()* methods. The task will block until one of the *get()* or *activate()* methods successfully completes.

This method can be used to fork off a second stream of references to the transaction descriptor stream. Note that the transaction descriptors themselves are not copied. The references returned by this method are referring to the same transaction descriptor instances obtained by the *get()* and *activate()* methods.

`function bit tee_mode(bit is_on);`

Turn the tee mode ON or OFF for this channel. Returns TRUE if the tee mode was previously ON. A threads blocked on a call to the *vmm_channel::tee()* method will not unblock execution if the tee mode is turned OFF. If the stream of references is not drained via the *vmm_channel::tee()* method, data will accumulate in the secondary channel when the tee mode is ON.

`function void connect(vmm_channel downstream);`

Connect the output of this channel instance to the input of the specified channel instance. The connection is performed with a blocking model (see section titled "In-Order Atomic Execution Model" on page 177) to communicate the status of the downstream channel to the producer interface of the upstream channel. Flushing this

channel will cause the downstream connected channel to be flushed as well. However, flushing the downstream channel will not flush this channel.

The effective full and empty levels of the combined channels is equal to the sum of their respective levels minus one. However, the detailed blocking behavior of the various interface methods will differ from using a single channel with an equivalent configuration. Additional zero-delay simulation cycles may be required while transaction descriptors are transferred from the upstream channel to the downstream channel.

Connected channels need not be of the same type but must carry compatible polymorphic data.

The connection of a channel into another one can be dynamically modified and broken by connection to a *`null`* reference. However, modifying the connection while there is data flowing through the channels may yield unpredictable behavior.

`function` *`class_name`* `for_each(bit reset = 0);`

Iterates over all of the transaction descriptors currently in the channel. The content of the active slot, if non-empty, is not included in the iteration. If the reset argument is TRUE, a reference to the first transaction descriptor in the channel is returned. Otherwise, a reference to the next transaction descriptor in the channel is returned. Returns *`null`* when the last transaction descriptor in the channel has been returned. It will keep returning *`null`* unless reset.

Modifying the content of the channel in the middle of an iteration will yield unexpected results.

`function int unsigned for_each_offset();`

Returns the offset of the last transaction descriptor returned by the *`vmm_channel::for_each()`* method. An offset of 0 indicates the first transaction descriptor in the channel.

`function bit record(string filename);`

Starts recording the flow of transaction descriptors added through the channel instance in the specified file. The *`vmm_data::save()`* method must be implemented for that transaction descriptor and defines the file format. A transaction descriptor is recorded when added to the channel by the *`vmm_channel::put()`* method.

A *null* filename stops the recording process. Returns TRUE if the specified file was successfully opened.

```
task bit playback(output bit success,
                  input string filename,
                  input vmm_data loader,
                  input bit metered = 0);
```

Locks all sources of the current channel and playback the transaction descriptors found in the specified file. The *vmm_data::load()* load method in the object specified by the *loader* argument is used and defines the file format. The transaction descriptors are added one by one in the order specified in the file. If the *metered* argument is TRUE, the transaction descriptors are added to the channel with the same relative time interval as they were originally put in when the file was recorded.

All consumers are locked out from the channel during playback. Normal operation resumes after the data has been entirely played back. Returns TRUE if the playback was successful.

VMM_BROADCAST

Channels are point-to-point data transfer mechanisms. If multiple consumers are extracting transaction descriptors from a channel, the transaction descriptors are distributed among the various consumers and each of the *N* consumers sees 1/*N* descriptors. If a point-to-multi-point mechanism is required, where all consumers must see all of the transaction descriptors in the stream, a *vmm_broadcast* component can be used to replicate the stream of transaction descriptors from a source channel to an arbitrary and dynamic number of output channels. If only two output channels are required, the *vmm_channel::tee()* method of the source channel may also be used.

Individual output channels can be configured to receive a copy of the reference to the source transaction descriptor (most efficient but the same descriptor instance is shared by the source and all like-configured output channels) or to use a new descriptor instance copied from the source object (least efficient but uses a separate instance that can be modified without affecting other channels or the original descriptor). A *vmm_broadcast* component can be configured to use references or copies in output channels by default.

In the *As Fast As Possible (AFAP)* mode, the full level of the output channels is ignored. Only the full level of the source channel will control the flow of data through the broadcaster. Output channels are kept non-empty as much as possible. As soon as an active output channel becomes empty, the next descriptor is removed from the source channel (if available) and added to all output channels, even if they are already full.

In the *As Late As Possible (ALAP)* mode, the slowest of the output or input channels controls the flow of data through the broadcaster. Only once *all* active output channels are empty, the next descriptor is removed from the source channel (if available) and added to all output channels.

If there are no active output channels, the input channel is continuously drained as transaction descriptors are added to it to avoid data accumulation.

This class is based on the *vmm_xactor* class.

vmm_log log;

Message service interface for this broadcaster. Set by the constructor and uses the name and instance name specified in the constructor.

```
function new( string name,
              string instance,
              vmm_channel source,
              bit use_references = 1,
              bcast_mode_typ mode = AFAP);
```

Creates a new instance of a channel broadcaster object with the specified name, instance name, source channel and broadcasting mode. If *use_references* is TRUE (i.e., non-zero), references to the original source transaction descriptors are assigned to output channels by default (unless individual output channels are configured otherwise).

See the documentation for the *broadcast_mode()* method on page 399 for a description of the available modes.

virtual function void start_xactor();

Starts this *vmm_broadcast* instance. The broadcaster can be stopped. Any extension of this method must call *super.start_xactor()*.

`virtual function void stop_xactor();`

Suspends this *`vmm_broadcast`* instance. The broadcaster can be restarted. Any extension of this method must call *`super.stop_xactor()`*.

`virtual function void`
`reset_xactor(reset_e rst_type = SOFT_RST);`

Resets this *`vmm_broadcast`* instance. The broadcaster can be restarted. The input channel and all output channels are flushed.

`virtual function void`
`broadcast_mode(bcast_mode_e mode);`

Changes the broadcasting mode to the specified mode. The new mode takes effect immediately. The available modes are specified by using one of the class-level enumerated symbolic values shown in Table A-6.

Table A-6. Broadcasting Mode Enumerated Values

Enumerated Value	Broadcasting Operation
`vmm_broadcast::ALAP`	***As Late As Possible***. Data is broadcast *only* when all active output channels are empty. This delay ensures that data is not broadcast any faster than the slowest of all consumers can digest it.
`vmm_broadcast::AFAP`	***As Fast As Possible.*** Active output channels are kept non-empty as much as possible. As soon as an active output channel becomes empty, the next descriptor from the input channel (if available) is immediately broadcast to all active output channels, regardless of their fill level. This mode *must not* be used if the data source can produce data at a higher rate than the slowest data consumer and if broadcast data in all output channels are not consumed at the same average rate.

```
virtual function int
          new_output(vmm_channel channel,
                     logic use_references = 1'bx);
```

Adds the specified channel instance as a new output channel to the broadcaster. If *use_references* is TRUE (i.e., non-zero), references to the original source transaction descriptor is added to the output channel. If FALSE (i.e., zero), a new instance copied from the original source descriptor is added to the output channel. If unknown (i.e., 1'bx), the default broadcaster configuration is used.

If there are no output channels, the data from the input channel is continuously drained to avoid data accumulation.

This method returns a unique identifier for the output channel that must be used to modify the configuration of the output channel.

Any user extension of this method must call *super.new_output()*.

```
virtual function void bcast_on(int unsigned output_id);
virtual function void bcast_off(int unsigned output_id);
```

Turns broadcasting to the specified output channel on or off. By default, broadcasting to an output channel is on. When broadcasting is turned off, the output channel is flushed and the addition of new transaction descriptors from the source channel is inhibited. The addition of descriptors from the source channel is resumed as soon as broadcasting is turned on.

If all output channels are off, the input channel is continuously drained to avoid data accumulation.

Any user extension of these methods should call *super.bcast_on()* or *super.bcast_off(),* respectively.

```
virtual protected function bit
                add_to_output(  int unsigned decision_id,
                                int unsigned output_id,
                                vmm_channel channel,
                                vmm_data obj);
```

Overloading this method allows the creation of broadcaster components with different broadcasting rules. If this function returns TRUE (i.e., non-zero), the transaction descriptor will be added to the specified output channel. If this function returns FALSE (i.e., zero), the descriptor is not added to the channel. If the output

channel is configured to use new descriptor instances, the *dat* parameter is a reference to that new instance.

This method is not necessarily invoked in increasing order of output identifiers. It is only called for output channels currently configured as ON. If this method returns FALSE for all output channels for a given broadcasting cycle, lock-up may occur. The *decision_id* argument is reset to 0 at the start of every broadcasting cycle and is incremented after each call to this method in the same cycle. It can be used to identify the start of broadcasting cycles.

If transaction descriptors are manually added to output channels, it is important that the *vmm_channel::sneak()* method be used to prevent the execution thread from blocking. It is also important that FALSE be returned to prevent that descriptor from being added to that output channel by the default broadcast operations and thus from being duplicated into the output channel.

The default implementation of this method always returns TRUE.

VMM_SCHEDULER

Channels are point-to-point transaction descriptor transfer mechanisms. If multiple sources are adding descriptors to a single channel, the descriptors are interleaved with the descriptors from the other sources in a fair but uncontrollable way. If a multi-point-to-point mechanism is required to follow a specific scheduling algorithm, a *vmm_scheduler* component can be used to identify which source stream should next be forwarded to the output stream.

This class is based on the *vmm_xactor* class.

vmm_log log;

Message service interface for this scheduler. Set by the constructor and uses the name and instance name specified in the constructor.

protected vmm_channel out_chan;

Reference to the output channel. Set by the constructor.

```
function new( string name,
              string instance,
```

```
                vmm_channel destination,
                int instance_id = -1);
```

Creates a new instance of a channel scheduler object with the specified name, instance name, destination channel and optional instance identifier.

`virtual function void start_xactor();`

Starts this *`vmm_scheduler`* instance. The scheduler can be stopped. Any extension of this method must call *`super.start_xactor()`*.

`virtual function void stop_xactor();`

Suspends this *`vmm_scheduler`* instance. The scheduler can be restarted. Any extension of this method must call *`super.stop_xactor()`*.

`virtual function void`
`reset_xactor(reset_e rst_typ = SOFT_RST);`

Resets this *`vmm_scheduler`* instance. The output channel and all input channels are flushed. If a *`HARD_RST`* reset type is specified, the scheduler election factory instance in the *`randomized_sched`* property is replaced with a new default instance.

`virtual function int new_source(vmm_channel chan);`

Adds the specified channel instance as a new input channel to the scheduler. This method returns an identifier for the input channel that must be used to modify the configuration of the input channel or -1 if an error occurred.

Any user extension of this method must call *`super.new_source()`*.

`virtual function void sched_on(int unsigned input_id);`
`virtual function void sched_off(int unsigned input_id);`

Turns scheduling from the specified input channel on or off. By default, scheduling from an input channel is on. When scheduling is turned off, the input channel is not flushed and the scheduling of new transaction descriptors from that source channel is inhibited. The scheduling of descriptors from that source channel is resumed as soon as scheduling is turned on.

Any user extension of this method should call *`super.sched_from_input()`* or *`super.sched_from_input()`*, respectively.

```
virtual protected task
                  schedule(output vmm_data obj,
                           input vmm_channel sources[$],
                           int unsigned input_ids[$]);
```

Overloading this method allows the creation of scheduling components with different rules. It is invoked for each scheduling cycle. The transaction descriptor returned by this method in the *obj* argument is added to the output channel. If this method returns *null*, no descriptor is added for this scheduling cycle. The input channels provided in the *sources* argument are all the currently non-empty ON input channels. Their corresponding input identifier is found in the *input_ids* argument.

New scheduling cycles are attempted whenever the output channel is not full. If no transaction descriptor is scheduled from any of the currently non-empty source channels, the next scheduling cycle will be delayed until an additional ON source channel becomes non-empty. If there are no empty input channels and no OFF channels, lock-up will occur.

The default implementation of this method randomizes the instance found in the *randomized_sched* property.

```
virtual protected task get_object(output vmm_data obj,
                                  vmm_channel source,
                                  int unsigned input_id,
                                  int offset);
```

This method is invoked by the default implementation of the *vmm_scheduler::schedule()* method to extract the next scheduled transaction descriptor from the specified input channel at the specified offset within the channel. Overloading this method allows access to or replacement of the descriptor that is about to be scheduled. User-defined extensions can be used to introduce errors by modifying the object, interfere with the scheduling algorithm by substituting a different object or recording of the schedule into a functional coverage model.

Any object that is returned by this method via the *obj* argument must either have been internally created or physically removed from the input source using the *vmm_channel::get()* method. If a reference to the object remains in the input channel (e.g., by using the *vmm_channel::peek()* or *vmm_channel::activate()* method), it is liable to be scheduled more than once as the mere presence of an instance in any of the input channel makes it available to the scheduler.

`vmm_scheduler_election randomized_sched;`

Factory instance randomized by the default implementation of the *`vmm_scheduler::schedule()`* method. Can be replaced with user-defined extensions to modify the election rules.

vmm_scheduler_election

This class implements the random election rules for the next scheduling cycle. The election is performed by randomizing an instance of this class. The default implementation provides a round-robin election process.

`int instance_id;`

Instance identifier of the *`vmm_scheduler`* class instance that is randomizing this object instance. Can be used to specified instance-specific constraints.

`int unsigned election_id;`

Incremented by the *`vmm_scheduler`* instance that is randomizing this object instance before every election cycle. Can be used to specified election-specific constraints.

`int unsigned n_sources;`

Number of sources. Equal to *`vmm_scheduler_election::sources.size()`*.

`vmm_channel sources[$];`

Input source channels with transaction descriptors available to be scheduled.

`int unsigned ids[$];`

Unique input identifiers corresponding to the source channels at the same index in the *`sources`* array.

`int unsigned id_history[$];`

A queue of the (up to) 10 last input identifiers that were elected.

`vmm_data obj_history[$];`

A list of the (up to) 10 last transaction descriptors that were elected.

`rand int unsigned source_idx;`

Index in the *`sources`* array of the elected source channel. An index of –1 indicates no election. The *`default_round_robin`* constraint block constrains this property to be in the *`0`* to *`sources.size()-1`* range.

`rand int unsigned obj_offset;`

Offset, within the source channel indicated by the *`source_idx`* property, of the elected transaction descriptor within the elected source channel. The *`default_round_robin`* constraint block constrains this property to be equal to 0.

`constraint default_round_robin;`

Constraints required by the default round-robin election process.

`function void post_randomize();`

The default implementation of this method helps performs the round-robin election.

VMM_NOTIFY

The *`vmm_notify`* class implements an interface to the notification service. The notification service provides a synchronization mechanism for concurrent threads or transactors. Unlike *`event`* variables, the operation of the notification is define at configuration time. Furthermore, notification can have status and timestamp information attached to their indication.

`function new(vmm_log log);`

Creates a new instance of this class, using the specified message service interface to issue error and debug messages.

`virtual function vmm_notify copy(vmm_notify to = null);`

Copies the current configuration of this notification service interface to the specified instance. If no instance is specified, a new one is allocated using the same message service interface as the original one. A reference to the target instance copied is returned.

Only the notification configuration information is copied and merged with any pre-configured notification in the destination instance. Copied notification configuration

will replace any pre-existing configuration for the same notification identifier. Status and timestamp information is **not** copied.

```
virtual function int
          configure( int notification_id = -1,
                     sync_e sync = ONE_SHOT);
```

Defines a new notification associated with the specified unique identifier. If a negative identifier value is specified, a new, unique identifier greater than 1,000,000 is returned. The thread synchronization mode of a notification is defined when the notification is configured, not when it is triggered or waited upon, using one of the *vmm_notify::ONE_SHOT*, *vmm_notify::BLAST*, or *vmm_notify::ON_OFF* synchronization types. This definition timing prevents a notification from being misused by the triggering or waiting threads.

Table A-7. Notification Synchronization Mode Enumerated Values

Enumerated Value	Broadcasting Operation
vmm_notify::ONE_SHOT	Only threads currently waiting for the notification to be indicated are notified.
vmm_notify::BLAST	All threads waiting for the notification to be indicated in the same timestep at the indication are notified. This mode eliminates certain types of race conditions.
vmm_notify::ON_OFF	The notification is level-sensitive. Notifications remain notified until explicitly reset. Threads waiting for a notification that is still notified will not wait. This mode eliminates certain types of race conditions.

A warning may be issued if a notification is configured more than once.

Notification identifiers numbered from 1,000,000 and up are reserved for automatically generated notification identifiers. Predefined notification identifiers in the VMM base classes use identifiers 999,999 and down. User-defined notification identifiers can thus use values 0 and up.

```
virtual function int is_configured(int notification_id);
```

Checks if the specified notification is currently configured. If this method returns 0, the notification is not configured. Otherwise, it returns an integer value corresponding

to the current *vmm_notify::ONE_SHOT*, *vmm_notify::ONE_BLAST* or *vmm_notify::ON_OFF* configuration.

virtual function bit is_on(int notification_id);

Checks if the specified *vmm_notify::ON_OFF* notification is currently in the *notify* state. If this method returns TRUE, the notification is in the notify state and any call to the *vmm_notify::wait_for()* method will not block. A warning is issued if this method is called on any other types of notifications.

virtual task wait_for(int notification_id);

Suspends the execution thread until the specified notification is notified. It is an error to specify an unconfigured notification. Use the *vmm_notify::status()* function to retrieve any status descriptor attached to the indicated notification.

virtual task wait_for_off(int notification_id);

Suspends the execution thread until the specified *vmm_notify::ON_OFF* notification is reset. It is an error to specify an unconfigured or a non-ON/OFF notification. The status returned by subsequent calls to the *vmm_notify::status()* function is undefined.

virtual function bit is_waited_for(int notification_id);

Checks if a thread is currently waiting for the specified notification, including waiting for an ON/OFF notification to be reset. It is an error to specify an unconfigured notification. The function returns TRUE if there is a thread known to be waiting for the specified notification.

Note that the knowledge about the number of threads waiting for a particular notification is not definitive and may be out of date. As threads call the *vmm_notify::wait_for()* method, the fact that they are waiting for the notification is recorded. Once the notification is indicated and each thread returns from the method call, the fact that they are no longer waiting is also recorded. But if the threads are externally terminated via the *disable* statement or a timeout, the fact that they are no longer waiting cannot be recorded. In this case, it is up to the terminated threads to report that they are no longer waiting by calling the *vmm_notify::terminated()* method.

When a notification is reset with a hard reset, no threads are assumed to be waiting for any notification.

```
virtual function void terminated(int notification_id);
```

Indicates to the notification service interface that a thread waiting for the specified notification has been disabled and is no longer waiting.

```
virtual function vmm_data status(int notification_id);
```

Returns the status descriptor associated with the specified notification when it was last indicated. It is an error to specify an unconfigured notification.

```
virtual function time timestamp(int notification_id);
```

Returns the simulation time when the specified notification was last indicated. It is an error to specify an unconfigured notification.

```
virtual function void indicate( int notification_id,
                                vmm_data status = null);
```

Indicates the specified notification with the optional status descriptor.

```
virtual function void
      set_notification( int notification_id,
                        vmm_notification ntfy = null);
```

Defines the specified notification using the specified notification descriptor. If the descriptor is *`null`*, the notification is undefined and can only be indicated using the *`vmm_notify::indicate()`* method. If a notification is already defined, the new definition replaces the previous definition.

```
virtual function vmm_notification
          get_notification(int notification_id);
```

Gets the notification descriptor associated with the specified notification, if any. If no notification descriptor is associated with the specified notification, *`null`* is returned.

```
virtual function void reset( int notification_id = -1,
                             reset_e rst_typ = SOFT);
```

Resets the specified notification. A *`vmm_notify::SOFT`* reset clears the specified *`ON_OFF`* notification and restarts the *`vmm_notification::indicate()`* and *`vmm_notification::reset()`* methods on any attached notification descriptor. A *`vmm_notify::HARD`* reset clears all status information and attached

notification descriptor on the specified event and further assumes that no threads are waiting for that notification. If no notification is specified, all notifications are reset.

Example A-9. Defining Three User-Defined Notifications

```
class bus_mon extends vmm_xactor;
   static int EVENT_A = 0;
   static int EVENT_B = 1;
   static int EVENT_C = 2;

   function new(...);
      super.new(...);
      super.notify.configure(this.EVENT_A);
      super.notify.configure(this.EVENT_B,
                             vmm_notify::ON_OFF);
      super.notify.configure(this.EVENT_C,
                             vmm_notify::BLAST);
   endfunction
endclass: bus_mon
```

vmm_notification

This class is used to describe a notification that can be autonomously indicated or reset based on a user-defined behavior, such as the composition of other notifications or external events. Notification descriptors are attached to notifications using the *`vmm_notify::set_notification()`* method.

`virtual task indicate(ref vmm_data status);`

Defines a method that, when it returns, causes the notification attached to the descriptor to be indicated. The value of the *`status`* argument is used as the indicated notification status descriptor. This method is automatically invoked by the notification service interface when a notification descriptor is attached to a notification using the *`vmm_notify::set_notification()`* method.

This method must be overloaded in user-defined class extensions. It can be used to implement arbitrary notification mechanisms, such as notifications based on a complex composition of other indications (e.g., notification expressions) or external events.

`virtual task reset();`

Defines a method that, when it returns, causes the ON/OFF notification attached to the notification descriptor to be reset. This method is automatically invoked by the notification service interface when a notification definition is attached to a *`vmm_notify::ON_OFF`* notification.

This method must be overloaded in user-defined class extensions.

Example A-10. Notification Indicated When Two Other Notifications Are Indicated

```
class notify_a_and_b extends vmm_notification;
   local vmm_notify notify;
   local int        a;
   local int        b;

   function new(vmm_notify notify,
                int        a,
                int        b) {
      this.notify = notify;
      this.a      = a;
      this.b      = b;
   }

   virtual task indicate(ref vmm_data status)
      fork
         void = this.notify.wait_for(a);
         void = this.notify.wait_for(b);
      join
   endtask
endclass: notify_a_and_b

class bus_mon extends vmm_xactor;

   static int EVENT_A = 0;
   static int EVENT_B = 1;
   static int EVENT_C = 2;

   function new(...);
      super.new(....);
      super.notify.configure(this.EVENT_A);
      super.notify.configure(this.EVENT_B,
                             vmm_notify::ON_OFF);
      super.notify.configure(this.EVENT_C,
                             vmm_notify::BLAST);

      begin
         notify_a_and_b AB = new(super.notify,
                                 this.EVENT_A,
                                 this.EVENT_B);
         super.notify.set_notification(this.EVENT_C,
                                       AB);
      end
   endfunction
endclass: bus_mon
```

VMM_XACTOR

This base class is to be used as the basis for all transactors, including bus-functional models, monitors and generators. It provides a standard control mechanism expected to be found in all transactors. The guidelines covering the development of transactors based on this class can be found in section titled "Transactors" on page 161.

```
function new( string name,
              string instance,
              int stream_id = -1);
```

Creates an instance of the transactor base class, with the specified name, instance name and optional stream identifier. The name and instance name are used to create the message service interface in the *vmm_xactor::log* property and the specified stream identifier is used to initialize the *vmm_xactor::stream_id* property.

```
virtual function string get_name();
virtual function string get_instance();
```

Returns the name and instance name of this transactor respectively.

```
vmm_log log;
```

Message service interface for messages issued from within this transactor instance.

```
int stream_id;
```

Unique identifier for the stream of transaction and data descriptors flowing through this transactor instance. It should be used to set the *vmm_data::stream_id* property of the descriptors as they are received or randomized by this transactor.

```
virtual function void
            prepend_callback(vmm_xactor_callbacks cb);
virtual function void
            append_callback(vmm_xactor_callbacks cb);
```

Prepends or appends the specified callback façade instance with this instance of the transactor. Callback methods will be invoked in the order in which they were registered.

A warning is issued if the same callback façade instance is registered more than once with the same transactor. A façade instance can be registered with more than one transactor. Callback façade instances can be unregistered and re-registered dynamically.

```
virtual function void
          unregister_callback(vmm_xactor_callbacks cb);
```

Unregisters the specified callback façade instance for this transactor instance. A warning is issued if the specified façade instance is not currently registered with the transactor. Callback façade instances can later be re-registered with the same or another transactor.

```
vmm_notify notify;
enum {XACTOR_IDLE;
      XACTOR_BUSY;
      XACTOR_STARTED;
      XACTOR_STOPPED;
      XACTOR_RESET};
```

Notification service interface and pre-configures notifications to indicate the state and state transitions of the transactor. The *vmm_xactor::XACTOR_IDLE* and *vmm_xactor::XACTOR_BUSY* notifications are *vmm_notify::ON_OFF*. All other events are *vmm_notify::ONE_SHOT*.

```
virtual function void start_xactor();
```

Starts the execution threads in this transactor instance. The transactor can later be stopped. Any extension of this method must call *super.start_xactor()*. The base class indicates the *vmm_xactor::XACTOR_STARTED* and *vmm_xactor::XACTOR_BUSY* notifications and resets the *vmm_xactor::XACTOR_IDLE* notification.

```
virtual function void stop_xactor();
```

Stops the execution threads in this transactor instance. The transactor can later be restarted. Any extension of this method must call *super.stop_xactor()*. The transactor will actually stop when the *vmm_xactor::wait_if_stopped()* or *vmm_xactor::wait_if_stopped_or_empty()* method is called. It is calls to these methods that define the granularity of stopping a transactor.

```
virtual function void
          reset_xactor(reset_e rst_typ = SOFT_RST);
```

Resets the state and terminates the execution threads in this transactor instance, according to the specified reset type. The base class indicates the *vmm_xactor::XACTOR_RESET* and *vmm_xactor::XACTOR_IDLE* notifications and resets the *vmm_xactor::XACTOR_BUSY* notification.

Table A-8. Reset Types

Enumerated Value	Broadcasting Operation
vmm_xactor::SOFT_RST	Clears the content of all channels, resets all ON_OFF notifications and terminates all execution threads but maintains the current configuration, notification service and random number generation state information. The transactor must be restarted. This reset type must be implemented.
vmm_xactor::PROTOCOL_RST	Equivalent to a reset signaled via the physical interface. The information affected by this reset is user defined.
vmm_xactor::FIRM_RST	Like *SOFT_RST*, but resets all notification service interface and random-number-generation state information. This reset type must be implemented.
vmm_xactor::HARD_RST	Resets the transactor to the same state found after construction. The registered callbacks are unregistered.

To facilitate the implementation of this method, the actual values associated with these symbolic properties are of increasing magnitude (e.g., *vmm_xactor::FIRM_RST* is greater than *vmm_xactor::SOFT_RST*). Not all reset types may be implemented by all transactors. Any extension of this method must call *super.reset_xactor(rst_type)* first to terminate the *vmm_xactor::main()* method, reset the notifications and reset the main thread seed according to the specified reset type. Calling *super.reset_xactor()* with a reset type of *vmm_xactor::PROTOCOL_RST* is functionally equivalent to *vmm_xactor::SOFT_RST.*

protected task wait_if_stopped()
protected task wait_if_stopped_or_empty(vmm_channel chan)

Blocks the thread execution if the transactor has been stopped via the *stop_xactor()* method or if the specified input channel is currently empty. These methods will indicate the *vmm_xactor::XACTOR_STOPPED* and *vmm_xactor::XACTOR_IDLE* notifications and reset the

vmm_xactor::XACTOR_BUSY notification. The tasks will return once the transactor has been restarted using the *start_xactor()* method and the specified input channel is not empty. These methods do not block if the transactor is not stopped and the specified input channel is not empty.

Calls to these methods define the granularity by which the transactor can be stopped without violating the protocol. If a transaction can be suspended in the middle of its execution, the *wait_if_stopped()* method should be called at every opportunity. If a transaction cannot be suspended, the *wait_if_stopped_or_empty()* method should only be called after the current transaction has been completed, before fetching the next transaction descriptor for the input channel.

Example A-11. Stopping a Transactor Execution at Appropriate Points

```
protected virtual task main();
   fork
      super.main();
   join none
   while (1) begin
      transaction tr;
      this.wait_if_stopped_or_empty(this.in_chan);
      this.in_chan.get(tr);
      ...
      this.wait_if_stopped();
      ...
   end
endtask: main
```

protected virtual task main();

This task is forked off whenever the *start_xactor()* method is called. It is terminated whenever the *reset_xactor()* method is called. The functionality of a user-defined transactor must be implemented in this method. Any additional subthreads must be started within this method, not in the constructor. It can have a blocking or non-blocking implementation.

Any extension of this method must first fork a call to *super.main()*.

virtual function void save_rng_state();

This method should save, in local properties, the state of all random generators associated with this transactor instance.

virtual function void restore_rng_state();

This method should restore, from local properties, the state of all random generators associated with this transactor instance.

virtual function void xactor_status(string prefix = "");

Displays the current status of the transactor instance in a human-readable format using the message service interface found in the *vmm_log::log* property, using *vmm_log::NOTE_TYP* messages. Each line of the status information is prefixed with the specified prefix.

‘vmm_callback(callback_class_name, method(args));

This macro simplifies the syntax of invoking callback methods in a transactor. For example, instead of:

```
foreach (this.callbacks[i]) begin
   ahb_master_callbacks cb;
   if ($cast_assign(cb, this.callbacks[i])) continue;
   cb.ptr_tr(this, tr, drop);
end
```

Use:

```
‘vmm_callback(ahb_master_callbacks,
              ptr_tr(this, tr, drop));
```

vmm_xactor_callbacks

This class implements a pure virtual base class for callback containments. See the documentation for the *vmm_xactor::append_callback()* method on page 411.

VMM_ATOMIC_GEN

A macro is used to define a class named <*class_name*>*_atomic_gen* for any user-specified class derived from *vmm_data*[1], using a process similar to the *‘vmm_channel* macro.

1. With a constructor callable without any arguments.

The atomic generator class is an extension of the *vmm_xactor* class and as such, inherits all of the public interface elements provided in the base class.

‘vmm_atomic_gen(class_name, “Class Description”);

Defines an atomic generator class named *<class_name>_atomic_gen* to generate instances of the specified class. The generated class must be derived from the *vmm_data* class and the *<class_name>_channel* class must exist.

```
function new( string instance,
              int stream_id = -1,
              <class_name>_channel out_chan = null);
```

Creates a new instance of the *<class_name>_atomic_gen* class with the specified instance name and optional stream identifier. The generator can be optionally connected to the specified output channel. If no output channel instance is specified, one will be created internally in the *<class_name>_atomic_gen::out_chan* property.

The name of the transactor is defined as the user-defined class description string specified in the class implementation macro appended with “*Atomic Generator*”.

<*class_name*>_channel out_chan;

References the output channel for the instances generated by this transactor. The output channel may have been specified via the constructor. If no output channel instances were specified, a new instance is automatically created. This reference in this property may be dynamically replaced but the generator should be stopped during the replacement.

int unsigned stop_after_n_insts;

The generator will stop after the specified number of object instances has been generated and consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own.

The default value of this property is 0.

<*class_name*> randomized_obj;

Transaction or data descriptor instance that is repeatedly randomized to create the random content of the output descriptor stream. The individual instances of the output

stream are *copied* from this instance, after randomization, using the *vmm_data::copy()* method.

The atomic generator uses a class factory pattern to generate the output stream instances. The generated stream can be constrained using various techniques on this property.

The *vmm_data::stream_id* property of this instance is set to the generator's stream identifier before each randomization. The *vmm_data::data_id* property of this instance is also set before each randomization. It will be reset to 0 when the generator is reset and after the specified maximum number of instances has been generated.

enum {GENERATED};

Notification identifier for the notification service interface in the *vmm_xactor::notify* property provided by the *vmm_xactor* base class. It is configured as a *vmm_xactor::ONE_SHOT* notification and is indicated immediately before an instance is added to the output channel. The generated instance is specified as the status of the notification.

enum {DONE};

Notification identifier for the notification service interface in the *vmm_xactor::notify* property provided by the *vmm_xactor* base class. It is configured as a *vmm_xactor::ON_OFF* notification and is indicated when the generator stops because the specified number of instances has been generated. No status information is specified.

virtual task inject(<*class_name*> data,
ref bit dropped);

Injects the specified transaction or data descriptor in the output stream. Unlike injecting the descriptor directly in the output channel, it counts toward the number of instances generated by this generator and will be subjected to the callback methods. The method returns once the instance has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.

<*class_name*>_atomic_gen_callbacks

This class implements a façade for atomic generator, transactor, callback methods. This class is automatically declared and implemented for any user-specified class by the atomic generator macro.

```
virtual task post_inst_gen(<class_name>_atomic_gen gen,
                           <class_name> data,
                           ref bit drop);
```

Callback method invoked by the generator after a new transaction or data descriptor has been created and randomized but before it is added to the output channel.

The *gen* argument refers to the generator instance that is invoking the callback method (in case the same callback extension instance is registered with more than one transactor instance). The *data* argument refers to the newly generated descriptor—which can be modified. If the value of the *drop* argument is set to non-zero, the generated descriptor will not be forwarded to the output channel, but the remaining registered callbacks will still be invoked.

VMM_SCENARIO_GEN

A macro is used to define a class named <*class_name*>*_scenario_gen* for any user-specified class derived from *vmm_data*[2], using a process similar to the *‘vmm_channel* macro.

The scenario generator class is an extension of the *vmm_xactor* class and as such, inherits all of the public interface elements provided in the base class.

```
‘vmm_scenario_gen( class_name, “Class Description”);
```

Defines a scenario generator class to generate sequences of related instances of the specified class. The specified class must be derived from the *vmm_data* class and the <*class_name*>*_channel* class must exist. It must also have a constructor with no arguments or that has default values for all of its arguments.

2. With a constructor callable without any arguments.

The macro defines classes named <*class_name*>*_scenario_gen*, <*class_name*>*_scenario*, <*class_name*>*_scenario_election* and <*class_name*>*_scenario_gen_callbacks*.

```
function new( string instance,
              int stream_id = -1,
              <class_name>_channel out_chan = null);
```

Creates a new instance of a scenario generator transactor with the specified instance name and optional stream identifier. The generator can be optionally connected to the specified output channel. If no output channel is specified, one will be created internally in the <*class_name*>*_scenario_gen::out_chan* property.

The name of the transactor is defined as the user-defined class description string specified in the class implementation macro appended with "*Scenario Generator*".

<*class_name*>_channel out_chan;

References the output channel for the instances generated by this transactor. The output channel may have been specified via the constructor. If no output channel was specified, a new instance is automatically created. The reference in this property may be dynamically replaced but the generator should be stopped during the replacement.

int unsigned stop_after_n_insts;

The generator will stop after the specified number of transaction or data descriptor instances have been generated and consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own based on the number of generated instances (but may still stop based on the number of generated scenarios).

The default value of this property is 0.

int unsigned stop_after_n_scenarios;

The generator will stop after the specified number of scenarios have been generated and entirely consumed by the output channel. The generator must be reset before it can be restarted. If the value of this property is 0, the generator will not stop on its own based on the number of generated scenarios (but may still stop based on the number of generated instances).

The default value of this property is 0.

<*class_name*>_scenario scenario_set[$];

Set of available scenario descriptors that may be repeatedly randomized to create the random content of the output stream. The *<class_name>_scenario_gen::select_scenario* property is used to determine which scenario descriptor, out of the available set of descriptors, is randomized next. The individual instances of the output stream are then created by calling the *<class_name>_scenario::apply()* method of the randomized scenario descriptor.

By default, this property contains one instance of the atomic scenario descriptor *<class_name>atomic_scenario*. Out of the box, the scenario generator will generate individual random descriptors.

The *vmm_data::stream_id* property of the randomized instance is assigned the value of the generator's stream identifier before randomization. The *vmm_data::scenario_id* property of the randomized instance is assigned a unique value before randomization. It will be reset to 0 when the generator is reset and after the specified number of instances or scenarios has been generated.

<*class_name*>_scenario_election select_scenario;

References the scenario descriptor selector that is repeatedly randomized to determine which scenario descriptor, out of the available set of scenario descriptors, will be randomized next.

By default, a round-robin selection process is used. The constraint blocks or randomized properties in this instance can be turned off or the instance can be replaced with a user-defined extension to modify the election rules.

enum {GENERATED};

Notification identifier for the *vmm_xactor::notify* notification service interface provided by the *vmm_xactor* base class. It is configured as a *vmm_notify::ONE_SHOT* notification and is indicated immediately before a scenario is applied to the output channel. The randomized scenario is specified as the status of the notification.

enum {DONE};

Notification identifier for the *vmm_xactor::notify* notification service interface provided by the *vmm_xactor* base class. It is configured as a *vmm_notify::ON_OFF* notification and is indicated when the generator stops

because the specified number of instances or scenarios has been generated. No status information is specified.

`virtual task inject_obj(<class_name> obj);`

Injects the specified descriptor in the output stream. Unlike injecting the descriptor directly in the output channel, it counts toward the number of instances and scenarios generated by this generator and will be subjected to the callback methods as an atomic scenario. The method returns once the descriptor has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.

`virtual task inject(<class_name>_scenario scenario);`

Injects the specified scenario descriptor in the output stream. Unlike injecting the descriptors directly in the output channel, it counts toward the number of instances and scenarios generated by this generator and will be subjected to the callback methods. The method returns once the scenario has been consumed by the output channel or it has been dropped by the callback methods.

This method can be used to inject directed stimulus while the generator is running (with unpredictable timing) or when the generated is stopped.

*<class_name>*_scenario

This class implements a base class for describing scenarios or sequences of transaction descriptors. This class named `<class_name>_scenario` is automatically declared and implemented for any user-specified class named "*class_name*" by the scenario generator macro, using a process similar to the `vmm_channel` macro.

`static vmm_log log;`

Message service interface to be used to issue generic messages when the message service interface of the scenario generator is not available or in scope.

`int stream_id;`

Stream identifier. It is set by the scenario generator before the scenario descriptor is randomized. Can be used to express stream-specific constraints.

`int scenario_id;`

Scenario identifier within the stream. It is set by the scenario generator before the scenario descriptor is randomized and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset or when the specified number of scenarios has been generated.

```
function int unsigned
                define_scenario(  string name,
                                  int unsigned max_len);
```

Defines a new scenario with the specified name and the specified maximum number of transactions or data descriptors. Returns a unique scenario identifier that should be assigned to an *`int unsigned`* property.

```
function void
   redefine_scenario( int unsigned scenario_kind,
                      string name,
                      int unsigned max_len);
```

Redefines the name and maximum number of descriptors in a previously defined scenario. Used to redefine an existing scenario instead of creating a new one and constraining the original scenario out of existence.

```
function string
            scenario_name(int unsigned scenario_kind);
```

Returns the name associated with the specified scenario identifier.

`rand int unsigned scenario_kind;`

When randomized, selects the identifier of the scenario that is generated. Constrained to the known scenario identifiers defined using the *`<class_name>_scenario::define_scenario()`* method. Can be constrained to modify the distribution of generated scenarios.

`rand int unsigned length;`

Randomized number of items in the scenario. Defines how many instances in the *`<class_name>_scenario::items[]`* property are part of the scenario.

`rand <class_name> items[];`

Instances of user-specified `<class_name>` that are randomized to form the scenarios. Only elements from index 0 to `<class_name>_scenario::length`-1 are part of the scenario.

The constraint blocks and `rand` attributes of the instances in the randomized array may be turned `ON` or `OFF` to modify the constraints on scenario items. They can also be replaced with extensions.

By default, the output stream is formed by *copying* the values of the items in this array onto the output channel.

`<class_name> using;`

Instance used in the default implementation of the `pre_randomize()` method when invoking the `fill_scenario()` method. Set to `null` by default. Can be replaced by an instance of a derived class to subject the items of the scenario to different constraints or content.

`rand int unsigned repeated;`

Number of times the items in the scenario are applied. The repeated instances in the scenario count toward the total number of instances generated but only one scenario is considered generated, regardless of the number of times it is repeated.

This property is unconstrained by default. To avoid accidentally repeating a scenario many times, a warning message will be issued if the value of this property is greater than the value specified in the `repeat_thresh` property.

`static int unsigned repeat_thresh;`

To avoid accidentally repeating a scenario many times because the `repeated` property was left unconstrained, a warning message will be issued if the value of the `repeated` property is greater than the value specified in this property. The default value is 100.

`function void`
`allocate_scenario(<class_name> using = null);`

Allocates a new set of instances in the `items` property, up to the maximum number of items in the maximum-length scenario. Any instance previously located in the `items` array is replaced. If a reference to an instance is specified in the `using`

argument, the array is filled by calling *vmm_data::copy()* on the specified instance. Otherwise, the array is filled with new instance of *<class_name>* class.

function void fill_scenario(*<class_name>* using = null);

Allocates new instances in the *items* property, up to the maximum number of items in the maximum-length scenario in any *null* element of the array. Any instance previously located in the *items* array is left untouched. If a reference to an instance is specified in the *using* argument, the array is filled by calling *vmm_data::copy()* on the specified instance. Otherwise, the array is filled with a new instance of *<class_name>* class.

virtual task apply(*<class_name>*_channel channel,
ref int unsigned n_insts);

Applies the items in the scenario descriptor to the specified output channel and returns when they have all been consumed by the channel. The *n_insts* argument is set to the number of instances that were consumed by the channel. By default, copies the values of the *items* array using their *vmm_data::copy()* method.

This method may be overloaded to define procedural scenarios.

*<class_name>*_atomic_scenario

This class implements a predefined atomic scenario descriptor. An atomic scenario is composed of a single unconstrained transaction or data descriptor. This class named *<class_name>_atomic_scenario* is automatically implemented for any user-specified class named "*class_name*" by the scenario generator macro, using a process similar to the *`vmm_channel* macro.

int unsigned ATOMIC;

Symbolic scenario identifier for the atomic scenario described by this descriptor. The atomic scenario is a single, random, unconstrained, transaction descriptor (i.e., an atomic descriptor).

constraint atomic_scenario;

Specifies the constraints of the atomic scenario. By default, the atomic scenario is a single unrepeated unconstrained item. This constraint block may be overridden to redefine the atomic scenario.

<*class_name*>_scenario_election

This class implements a random selection process for selecting the next scenario descriptor, from a set of available descriptors, to be randomized next. This class named `<class_name>_scenario_election` is automatically implemented for any user-specified class named "*class_name*" by the scenario generator macros, using a process similar to the `‘vmm_channel` macro.

`int stream_id;`

Stream identifier. It is set by the scenario generator to the value of the generator stream identifier before the scenario selector is randomized. Can be used to express stream-specific constraints.

`int scenario_id;`

Scenario identifier within the stream. It is set by the scenario generator before the scenario selector is randomized and incremented after each randomization. Can be used to express scenario-specific constraints. The scenario identifier is reset to 0 when the scenario generator is reset or when the specified number of scenarios has been generated.

`int unsigned n_scenarios;`

Number of available scenario descriptors in the scenario set. The final value of the `select` property must be in the `[0:n_scenarios-1]` range.

`int unsigned last_selected[$];`

A history (maximum of 10) of the last scenario selections. Can be used to express constraints based on the historical distribution of the selected scenarios (e.g., "never select the same scenario twice in a row").

`int unsigned next_in_set;`

The next scenario descriptor index that would be selected in a round-robin selection process. Used by the `round_robin` constraint block.

`<class_name>_scenario scenario_set[$];`

The available set of scenario descriptors. Can be used to procedurally determine which scenario to select or to express constraints based on the scenario descriptors.

`rand int select;`

The index, within the *`scenario_set`* array, of the selected scenario descriptor to be randomized next.

`constraint round_robin;`

Constrains the random scenario selection process to a round-robin selection. This constraint block may be turned off to produce a random scenario selection process or allow a different constraint block to define a different scenario selection process.

<*class_name*>_scenario_gen_callbacks

This class implements a façade for callback containments for the scenario generator transactor. This class named *`<class_name>_scenario_gen_callbacks`* is automatically implemented for any user-specified class named "*class_name*" by the scenario generator macro, using a process similar to the *`‘vmm_channel`* macro.

```
virtual task pre_scenario_randomize(
                    <class_name>_scenario_gen gen,
                    ref <class_name>_scenario scenario);
```

Callback method invoked by the generator after a new scenario has been selected but before it is randomized. The *`gen`* argument refers to the generator instance that is invoking the callback method. The *`scenario`* argument refers to the newly selected scenario descriptor which can be modified. Note that any modifications of the randomization state of the scenario descriptor—such as turning constraint blocks ON or OFF—will remain in effect the next time the scenario descriptor is selected to be randomized. If the reference to the scenario descriptor is set to *`null`*, the scenario will not be randomized and a new scenario will be selected.

To minimize memory allocation and collection, it is possible that the elements of the scenarios may not be allocated. Use the *`<class_name>_scenario::allocate_scenario()`* or *`<class_name>_scenario::fill_scenario()`* to allocate the elements of the scenario if necessary.

```
virtual task post_scenario_gen(
                        <class_name>_scenario_gen gen,
                        <class_name>_scenario scenario,
                        ref bit dropped);
```

Callback method invoked by the generator after a new scenario has been randomized but before it is applied to the output channel. The *`gen`* argument refers to the

generator instance that is invoking the callback method. The *scenario* argument refers to the newly randomized scenario that can be modified. Note that any modifications of the randomization state of the scenario descriptor—such as turning constraint blocks ON or OFF—will remain in effect the next time the scenario descriptor is selected to be randomized. If the value of the *dropped* argument is set to non-zero, the generated instance will not be applied to the output channel.

APPENDIX B VMM CHECKER LIBRARY

This appendix describes the checkers currently available in the VMM Checker Library. In the first group, there are 31 checkers that are equivalent to the checkers in the Accellera OVL but contain extensions for coverage. A second group of 19 checkers verify more complex behaviors than those in the first group.

Assertions are enabled globally by defining the symbol `ASSERT_ON`. If this symbol is not defined, the code for all checkers is physically removed at compile-time.

All the checkers contain coverage statements that can be globally enabled by defining the `COVER_ON` symbol. In addition, three coverage levels can be independently enabled on a per-instance basis. Level 1 coverage provides an indication of the coverage of the trigger conditions of the checker and, in some cases, of the basic functionality. Level 2 coverage collects data on the profiles of delay or data values observed during the simulation. Finally, Level 3 coverage provides information on the occurrence of corner cases such as hitting the user-specified minimum and maximum values on delays and value ranges.

OVL-EQUIVALENT CHECKERS (SVL)

This section describes the 31 OVL-like checkers. All checkers, except `assert_proposition`, are triggered at the positive edge of a triggering signal or expression `clk`. A *clock cycle* is defined as the duration between two consecutive positive edges of the clock signal.

The values of all actual signals or expressions in the ports of the checkers are sampled just before the positive edge of `clk`. Therefore, any pulses happening on the signals or expressions between consecutive positive edges of `clk` are not observed by the checkers.

Moreover, whenever an edge of a signal or expression on a port of a checker other than the clock is used in the checker, it is the sampled from the edge, as detected by looking at two consecutive samples of the signal.

The checker `assert_proposition` monitors an expression at all times but fires only when the `test_expr` undergoes a falling transition (from `1` or `x` or `z` to `0` (false)).

In the following descriptions, an assertion that *fires* means that an error condition has been detected.

assert_always — Continuously monitors `test_expr` at every positive edge of clock signal `clk`. `test_expr` must always evaluate to *true*. If `test_expr` evaluates to *false*, the assertion fires. The `test_expr` can be any valid expression.

assert_always_on_edge — Continuously monitors the `test_expr` at every specified edge of the `sampling_event`. `test_expr` must always evaluate *true* at the `sampling_event`. If `test_expr` evaluates to *false*, the assertion fires. Note that the transition on the sampling event is determined by sampling `sampling_event` at two consecutive positive edges of the clock signal `clk`.

assert_change — Continuously monitors the `start_event` at every positive edge of the clock signal `clk`. When `start_event` is *true*, the checker ensures that the expression, `test_expr` changes values on a clock edge at some point within the next `num_cks` number of clocks.

assert_cycle_sequence — Verifies the following conditions:

- When `necessary_condition` = `0`, if all `num_cks-1` first bits of a vector of Boolean events (`event_sequence[num_cks-1:1]`) are *true (1)* in consecutive clock cycles, the last Boolean (`event_sequence[0]`) must be *true* in the next clock cycle.
- When `necessary condition = 1`, if the first bit of a vector of (`event_sequence[num_cks-1]`) is *true*, then all the remaining `event_sequence[num_cks-2:0]` bits must become true in the subsequent `num_cks-1` clock cycles.

assert_decrement — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. It checks that `test_expr` always decreases by the value specified by `value`. The `test_expr` can be any valid expression. The checker will not start until the first clock edge after `reset_n` is asserted.

assert_delta — Continuously monitors `test_expr` at every positive edge of clock signal `clk`. It verifies that `test_expr` always changes value by a value greater than or equal to `min` and less than or equal to `max` value. The `test_expr` can be any valid expression. The checker will not start until the first clock edge after `reset_n` is asserted.

assert_even_parity — Ensures that the variable, `test_expr`, has an even number of bits set to `1` at any positive edge of the clock signal `clk`.

assert_fifo_index — Ensures that a FIFO element a) never overflows or underflows b) allows/disallows simultaneous push and pop operations.

assert_frame — Validates proper cycle timing relationships between two events in the design. When a `start_event` (a bit) evaluates *true*, then `test_expr` must evaluate *true* within a `minimum` and `maximum` number of clock cycles.

assert_handshake — Continuously monitors the `req` and `ack` signals at every positive edge of the clock signal `clk`. It ensures that `ack` occurs after `req` within a specified `minimum` and `maximum` number of clock cycles. Both `req` and `ack` must go inactive prior to starting a new cycle. Verifying that `req` is persistent until `ack` arrives and that it remains active for some cycle after `ack` is controlled by checker parameters.

assert_implication — Continuously monitors `antecedent_expr`. If it evaluates to *true*, then it verifies that the `consequent_expr` is *true*. When `antecedent_expr` evaluates to *false*, then `consequent_expr` expression will not be checked at all and the implication is satisfied.

assert_increment — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. It verifies that `test_expr` increases by the value specified by `value`. The `test_expr` can be any valid expression. The checker will not start until the first clock edge after `reset_n` is asserted.

assert_never — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. It verifies that `test_expr` never evaluates *true*. The `test_expr` can be any valid expression. When `test_expr` evaluates *true*, this checker fires.

assert_next — Validates proper cycle timing relationships between two events in the design. When a `start_event` evaluates *true*, then the `test_expr` must evaluate *true* exactly `num_cks` number of clock cycles later. This checker supports overlapping sequences.

assert_no_overflow — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. It verifies that a specified `test_expr` will never:

- Change value from a `max` value (default is `(2**width)-1`) to a value greater than `max`, or
- Change value from a `max` value (default is (`2**width)-1`) to a value less than or equal to a `min` value (default is 0).

assert_no_transition — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. When it evaluates to the value of `start_state`, it ensures that `test_expr` will never transition to the value of `next_state`. The `width` parameter defines the number of bits in `test_expr`.

assert_no_underflow — Continuously monitors `test_expr` at every positive edge of the clock signal `clk`. This checker verifies that `test_expr` will never:

- Change value from a `min` value (default is `0`) to a value less than `min`, or
- Change to a value greater than or equal to `max` (default is `(2**width)-1`).

assert_odd_parity — Ensures that the variable, `test_expr`, has an odd number of bits set to `1` at any positive edge of the clock signal `clk`.

assert_one_cold — Ensures that the variable, `test_expr`, has only one bit set to `0` at any positive clock edge when the checker is configured for no inactive states. The checker can also be configured to accept all bits equal to either `0` or `1` as the inactive level.

assert_one_hot — Ensures that the variable, `test_expr`, has only one bit set to `1` at any positive edge of the clock signal `clk`.

assert_proposition — Continuously monitors `test_expr` and verifies that `test_expr` always evaluate *true*. If `test_expr` transits from *true* to *false* while `reset_n` is `1`, the checker fires. Unlike `assert_always`, `test_expr` is not sampled by a clock.

assert_quiescent_state — continuously monitors *state_expr* at every positive edge of the sampling event *sample_event* and verifies that the value *state_expr* is equal to the value *check_value* and optionally at the end of simulation.

assert_range — Continuously monitors *test_expr* at every positive edge of the clock signal *clk*. The checker ensures that the *test_expr* is always within the *min* and *max* value range.

assert_time — Continuously monitors *start_expr*. When it evaluates *true*, the checker ensures that *test_expr* evaluates to *true* for the next *num_cks* number of clock cycles.

assert_transition — Continuously monitors *test_expr* at every positive edge of the clock signal *clk*. When *test_expr* evaluates to the value *start_state*, the checker ensures that *test_expr* will always change to the value of *next_state*. The *width* parameter defines the number of bits in *test_expr*.

assert_unchange — Continuously monitors *start_event* at every positive edge of the clock signal *clk*. When *start_event* evaluates *true*, the checker ensures that *test_expr* will not change value within the next *num_cks* number of clock cycles.

assert_width — Continuously monitors *test_expr*. When *test_expr* evaluates *true*, it ensures that *test_expr* evaluates to *true* for a specified *minimum* number of clock cycles and does not exceed a *maximum* number of clock cycles.

assert_win_change — Continuously monitors *start_event* at every positive edge of the clock signal *clk*. When *start_event* evaluates *true*, it ensures that *test_expr* changes values prior to and including the occurrence of *end_event*.

assert_win_unchange — Continuously monitors *start_event* at every positive edge of the clock signal *clk*. When *start_event* evaluates *true*, it ensures that *test_expr* will not change in value up to and including *end_event* becoming *true*.

assert_window — Continuously monitors *start_event* at every positive edge of the clock signal *clk*. When *start_event* evaluates *true*, it ensures that the *test_expr* evaluates *true* at every successive positive clock edge of *clk* up to and including the *end_event* expression becoming *true*. This checker does not evaluate

test_expr on *start_event.* It begins evaluating *test_expr* at the next positive clock edge of *clk*.

assert_zero_one_hot — Continuously monitors *test_expr* at every positive edge of the clock signal *clk*. It verifies that *test_expr* has exactly one bit asserted or no bit asserted.

ADVANCED CHECKERS

This section describes 19 advanced checkers. These advanced checkers use the same controls as the OVL-equivalent checkers described in the previous section. In addition, they have a clock edge selection parameter, *edge_expr*, that lets the user select *posedge* or *negedge* clock edge selection for sampling in the assertions and cover statements.

assert_arbiter — Ensures that a resource arbiter provides grants to corresponding requests within *min_lat* and *max_lat* cycles. *reqs* and *grants* are vectors of size *[no_chnl-1:0]* where the bits correspond to the individual channels. They are assumed to be *1* when active. The checker can verify a priority arbitration scheme alone or in conjunction with (as a secondary criterion) round-robin, FIFO or LRU selection algorithms. The checks are not enabled unless *reset_n* evaluates *true*.

assert_bits — Ensures that the value of *exp* has between *min* and *max* number of bits that are asserted or deasserted as indicated by the *deasserted* flag. The check is not enabled unless *reset_n* evaluates *true*.

assert_code_distance — Ensures that when *exp* changes, the number of bits that are different compared to *exp2*—the Hamming distance—are at least *min* but no more than *max* in number. The check is not enabled unless *reset_n* evaluates *true*.

assert_data_used — Ensures that data from *src[sleft:sright]* appears in *dest[dleft:dright]* within the window specified as *start* cycles from after the time *trigger* is asserted until *finish* number of cycles after *trigger* is asserted.

assert_driven — Ensures that all bits of *exp* are driven (i.e., none are '*Z*' or '*X*'). The check is not enabled unless *reset_n* evaluates *true*.

assert_dual_clk_fifo — Checker for a dual-clock, single-input and single-output FIFO. It assumes that enqueuing is enabled when *enq* is asserted at the active clock edge of *enq_clk* and effectively occurs *enq_lat* cycles later. Dequeuing is enabled when *deq* is asserted at the active edge of *deq_clk* and effectively occurs *deq_lat* cycles later. It can verify that neither overflow or underflow of the FIFO occurs, that it reaches a watermark and that the enqueued data value is the correct one upon dequeue.

assert_fifo — Checker for a single-clock, single-input and single-output FIFO. All signals are sampled at the active edge of the clock signal *clk*. It assumes that enqueuing is enabled when *enq* is asserted and effectively occurs *enq_lat* cycles later. Dequeuing is enabled when *deq* is asserted and effectively occurs *deq_lat* cycles later. It can verify that neither overflow or underflow of the FIFO occurs, that it reaches a watermark and that the enqueued data value is the correct one upon dequeue. Also, if *pass_thru* is *1*, it allows simultaneous enqueue and dequeue of data on empty or full queue. Otherwise a dequeue on an empty queue will report an underflow.

assert_hold_value — Ensures that *exp* of width *bw* remains at *value* for *min* to *max* number of cycles. That is, it must stay at *value* for *min* cycles, then it may change and after *max* cycles it must change to some other value. The check is not enabled unless *reset_n* evaluates *true*.

assert_memory_async — Ensures the integrity of an asynchronous memory content and access. When *addr_chk* evaluates *true*, it ensures that *start_addr* <= *raddr* <= *end_addr* as sampled by the *negedge* of *ren*, and that *start_addr* <= *waddr* <= *end_addr* as sampled by the *negedge* of *wen*. All other checks apply only if the address is valid. There is no clock other than the *ren* and *wen* expressions that indicate when each operation is to take place by their falling edges.

Checks can also be enabled to verify that memory locations are written into before being read, that there is at least one read between two consecutive writes to an address, or similarly that there is at least one write between two consecutive reads to an address. The checker can also verify that the value written last to a memory location is the one being read out later.

assert_memory_sync — Ensures the integrity of a synchronous memory content and access. When *addr_chk* evaluates true, it ensures that *start_addr* <= *raddr* <= *end_addr* when *ren* is true as sampled by the active edge of *rclk*, and that *start_addr* <= *waddr* <= *end_addr* when *wen* is true at the active edge of *wclk*. All other checks apply only if the address is valid.

Checks can also be enabled to verify that memory locations are written into before being read, that there is at least one read between two consecutive writes to an address, or similarly that there is at least one write between two consecutive reads to an address. The occurrence of simultaneous read and write operation when `rclk` is the same as `wclk` can be verified. The checker can also verify that the value written last to a memory location is the one being read out later or at the same time if `pass_thru` is enabled.

assert_multiport_fifo — Checker for a single-clock, multi-input and multi-output FIFO. `enq` and `deq` are bit vectors of equal size `no_ports`. Each pair of corresponding bits in these vectors defines the enqueue and dequeue enable signals for a FIFO port. Bit `0` has the lowest priority, while the highest-order bit `no_ports-1` has the highest priority. The enqueue port and the dequeue port of the highest priority are processed at every active `clk` edge.

`enq_data` is a concatenation of the data from the different ports, dimensioned as `[no_ports*elem_size-1:0]`, with data vectors appearing in the same order as the `enq` requests. Whenever a bit in `enq` is asserted `1`, the corresponding data port in `enq_data` must be valid after `enq_lat` clock cycles. Only the highest-priority data is actually enqueued.

`deq_data` is a concatenation of the data from the different ports. It is assumed that it is dimensioned the same way as `enq_data`, with data vectors appearing in the same order as the `deq` requests. Whenever a bit in `deq` is asserted `1`, the corresponding data port in `deq_data` must be valid after `deq_lat` clock cycles. Only the data of the highest-priority dequeue request is compared with the reference data when `value_chk` is `1`. Overflow, underflow, watermark, value and pass-thru checks can be enabled as in the `assert_fifo` checker.

assert_mutex — Ensures that `a` and `b` never evaluate *true* at the same time. The checker is not enabled unless `reset_n` evaluates *true*.

assert_next_state — Ensures that, when `exp` is in current state `cs`, `exp` will transition to one of the specified legal next states in `ns`. `no_ns` specifies the number of legal next states. `ns` is a bit vector of the concatenated legal state values that `exp` can transition to from `cs`.

assert_no_contention — Ensures that `bus` always has a single active driver and that there is no '`X`' or '`Z`' on the bus when driven (`en_vector != 0`). The total number of `en_vector` bits that are asserted can be at most 1. `min_quiet` and `max_quiet` define and interval in the number of clock cycles within when the bus may remain quiet, i.e., no diver enabled.

assert_reg_loaded — Ensures that the register *dst_reg* is loaded with *src* data. The check for *dst_reg* holding the memorized value of *src* starts with *delay* cycles (minimum 1, which is default) after the *trigger* condition evaluates *true* and within *end_cycle* cycles after the *trigger* evaluates *true* or when *stop* becomes *true* (whichever occurs first).

assert_req_ack_unique — Verifies that each *req* receives an *ack* within the specified interval *min_time* and *max_time* active clock edges of *clk*. The arriving *ack*'s are attributed to *req*'s in a FIFO order.

assert_stack — Verifies operations of a stack. When *push* is asserted *1*, it ensures that there is no stack overflow. *push_lat* specifies the number of clock cycles between the assertion of *push* and when *push_data* is valid. Similarly, when *pop* is asserted *1,* it ensures that the stack is not empty. *pop_lat* specifies the number of clock cycles between the assertion of *pop* and when *pop_data* must be valid. Data value, stack empty, full, watermark and pass-thru checks can be selectively enabled.

assert_valid_id — The signal *issued_sig* asserted *1* validates a request identified by the value in *issued_id*. This *request* is expected to be acknowledged by *ret_id* validated by *ret_sig* asserted *1* within *[min_lat:max_lat]* latency. A *reset_sig* asserted *true* with *reset_id* value of one of the currently issued and still outstanding IDs resets that outstanding ID to empty, i.e., a *ret_sig* asserted for the ID is then considered as invalid until newly issued.

The bit width *id_bw* of the IDs can be any value supported by the tool; however, the maximum number of outstanding IDs at any time is limited by the value of the parameter *max_ids*. For a given ID, there can be at most *max_out_per_id* outstanding issues. The arriving returns of that ID are matched in a FIFO manner to the requests when verifying the latency of the return (similarly as in the *assert_req_ack_unique* checker).

assert_value — Ensures that *exp* can only be one of the specified values in a set. *no_vals* indicates the number of values in the set, which is defined by a bit vector *vals* of width *[bw*no_vals-1 : 0]* of the concatenated values of *bw* bits each that *exp* must evaluate to.

APPENDIX C XVC STANDARD LIBRARY SPECIFICATION

This appendix specifies the detailed behavior of a set of base and utility classes that can be used to implement an XVC-compliant verification environment and verification components. The actual implementation of these classes is left to each tool provider.

XVC_MANAGER

This class is a base class for implementing XVC management functions, as described in “XVC Manager” on page 316. A predefined XVC manager, as described in “Predefined XVC Manager” on page 317 is specified in section titled “vmm_xvc_manager” on page 444.

`vmm_log log;`

Message service interface used to issue all messages from the XVC manager. The name is specified as “XVC Manager” and the instance name is the instance name of the *`xvc_manager`* instance, as specified in the constructor.

`vmm_log trace;`

Message service interface for execution trace messages that may be routed differently than the generic messages issued through the message service instance in the *`log`* class property.

`vmm_notify notify;`

Notification service interface for the global notifications. The event identifier is the same as the global notification identifier specified in the test file. All events corresponding to global notifications are triggered *`ONE_SHOT`*.

`function new(string instance = "Main");`

Creates an instance of the XVC manager with the specified instance name.

`function bit add_xvc(xvc_xactor xvc);`
`function bit remove_xvc(xvc_xactor xvc);`

Puts or removes the specified XVC instance under the control of this XVC manager. Returns non-zero if the operation is successful and error-free. An XVC instance cannot be under the control of more than one XVC manager at any given time.

XVC instances can only be added or removed from the control of an XVC manager when a manager is not running a test.

`function bit split( string command,`
`ref string argv[]);`

Splits the specified command into blank-separated tokens, suitable for the *`xvc_xactor::parse()`* method. Quotes and escaped characters are interpreted like the C shell when splitting arguments into main's *`argv`* array.

`protected xvc_xactor xvcQ[$];`

Array of XVC instances under the control of this XVC manager instance. The content of this array is managed using the *`add_xvc()`* and *`remove_xvc()`* methods.

XVC_XACTOR

This class is a base class for implementing XVC-compliant transactors, as described in "Extensible Verification Components" on page 306. This base class is derived from the *`vmm_xactor`* class and offers the following additional interface elements:

`vmm_log trace;`

Message service interface for execution trace messages that may be routed differently than the generic messages issued through the message service instance in the *`vmm_xactor::log`* class property.

vmm_notify notify;

Notification service interface for the local notifications. The event identifier is the same as the local notification identifier specified in the test file. All events corresponding to local notifications are triggered *ONE_SHOT*.

```
function new( string name,
              string instance,
              int stream_id = -1,
              xvc_action_channel action_chan = null,
              xvc_action_channel interrupt_chan = null);
```

Creates an instance of the XVC transactor with the specified name and instance name and optional stream identifier. The input action and interrupt channels are optionally connected to the specified channel instances. Action and interrupt channels, if specified, are reconfigured to a full level of 1 and 64 k respectively. The name, instance name and stream identifier are used as the *vmm_xactor* name, instance name and stream identifier, respectively. The name and instance name will be used to configure the *vmm_log* instance found in the *vmm_xactor* base class.

function void add_action(xvc_action action);

Adds the specified XVC action descriptor to the known actions of the XVC transactor. New action definitions may hide previous definitions as they are considered by the *parse()* method in the reverse order of registration.

function xvc_action parse(string argv[]);

Parses the specified action command and returns the corresponding action descriptor. If the action is not known to the XVC transactor, *null* is returned. The command is specified as an array of string tokens similar to *argv* in C's *main()* function argument.

xvc_action_channel action_chan;

Input channel for actions to be executed by the XVC transactor. If no channel instance was specified in the constructor, a new instance is internally allocated. The XVC transactor uses an in-order, blocking completion model, as described in section titled "In-Order Atomic Execution Model" on page 177.

`xvc_action_channel interrupt_chan;`

Input channel for interrupt actions to be executed by the XVC transactor at the earliest opportunity. If no channel instance was specified in the constructor, a new instance is internally allocated. An interrupt action will be executed after the current action execution completes of when the current action invokes the *`wait_if_interrupted()`* method. The XVC transactor uses an in-order, non-blocking completion model for interrupt actions, as described in section titled "Out-of-Order Atomic Execution Model" on page 182.

`protected task wait_if_interrupted();`

Suspends the execution thread if an interrupt action is waiting to be executed by the XVC. This method must only be called from within an implementation of the *`xvc_action::execute()`* method.

`protected vmm_channel exec_chan;`

Channel that must be used to execute the actions in the XVC.

`protected vmm_xactor xactors[];`

Lower-level transactors used by this XVC. Actions may require the registration of callback extensions to implement their execution.

XVC_ACTION

This class is a base class to implement XVC action descriptors. An action descriptor defines the command used to invoke it and how to execute it. Actions are XVC-specific and cannot be executed on different XVCs. This base class is derived from the *`vmm_data`* class and offers the following additional interface elements:

```
function new( string name,
              vmm_log log);
```

Creates a new instance of an action descriptor. The action is named using the specified name and the specified message interface is passed to the *`vmm_log::new()`* method.

`function string get_name();`

Returns the name of the action, as specified in the constructor.

virtual function xvc_action parse(string argv[]);

Parses the specified command and returns a new instance of the action descriptor that corresponds action descriptor. Returns *null* without issuing any error or warning messages if the command is not recognized.

virtual task execute(vmm_channel exec_chan,
xvc_xactor xvc);

Executes the action described by this instance of the action descriptor, through the specified input channel. The action is executed by generating and putting the necessary transaction descriptor in the specified input channel.

At appropriate points during the execution of the action, the *xvc_xactor::wait_if_interrupted()* method should be called to let interrupt actions be executed. The calls to this method define the granularity of the action execution. For example, an atomic action would never call *xvc_xactor::wait_if_interrupted()*.

vmm_xactor_callbacks callbacks[];

Transactor callbacks extensions that must be registered with the transactors in the XVC to properly execute the action described by this instance of the action descriptor, prior to invoking the *execute()* method. If not *null*, the callback extension instance is prepended to the registered callbacks of the corresponding lower-level transactor in the XVC before the action is executed, then unregistered upon completion of the execution.

virtual function int unsigned
byte_pack(ref logic [7:0] bytes[],
input int unsigned offset = 0,
input int kind = -1);

The default implementation packs the name of the action descriptor into the specified dynamic array of bytes, starting at the specified offset in the array and ending with a byte set to *8'h00*. The array is resized appropriately. Returns the number of bytes added to the array.

```
virtual function int unsigned
              byte_unpack( const ref logic [7:0] bytes[],
                           input int unsigned offset = 0,
                           input int len = -1,
                           input int kind = -1);
```

The default implementation unpacks the name of the action descriptor from the specified offset in the specified dynamic array until a byte set to *8'h00*, the specified number of bytes have been unpacked or the end of the array is encountered, whichever comes first. Returns the number of bytes unpacked.

```
virtual function int unsigned byte_size(int kind = -1);
```

The default implementation returns the length of the action descriptor name plus one.

```
virtual function int unsigned
                        max_byte_size(int kind = -1);
```

The default implementation returns the length of the action descriptor name plus one.

VMM_XVC_MANAGER

The class implements the predefined XVC manager as described in "Predefined XVC Manager" on page 317. It is implemented as an extension of the *xvc_manager* base class and provides the following additional elements.

```
task run(string testfile);
```

Starts all of the XVCs under the control of the manager and runs the test in the specified command file on the XVC manager instance. This task returns once the test has completed as defined in the test itself. The XVCs are not stopped nor reset when the test completes and are still running.

Notifications

The XVC manager base class provides a notification service interface in its *xvc_manager::notify* class property. The predefined XVC manager uses notifications to coordinate actions and XVCs. XVCs also use notifications in their respective notification service interface in their *vmm_xactor::notify* class property to coordinate with the predefined XVC manager.

Events are used in a scenario description in place of decision-making constructs. This is partly to reduce the complexity of the scenario description syntax, but more importantly to keep the scenario description as a portable top-level entity. Any specific or complex decision-making logic can easily be implemented as an XVC action, which in turn could indicate more events.

Event indications can be used to control the execution of a scenario. Users may specify a trigger event for any action within a scenario description. The predefined XVC manager will wait for that event to be indicated before causing an XVC to execute the corresponding action. For example, the XVC manager can instruct XVC "A" to execute actions A, B and C; then wait for event 1 to be indicated before executing action D.

Scenario events are defined as local to a test scenario or global to all scenarios. A scenario event is mapped to a *`vmm_notify::ONE_SHOT`* notification with the same numerical identifier in the predefined XVC manager notification service interface and in all XVC notification service interfaces. When a scenario event is indicated, the corresponding notification is indicated in the XVC manager and in all XVC instances.

XVC notifications can be mapped onto scenario events using the *`MAPEVENT`* command. When an XVC indicates a notification mapped to a scenario event, the indication will be propagated to the XVC manager and all other XVCs.

A combination of scenario events can be mapped onto a single scenario event using the *`MAP`* command.

The remainder of this section describes the syntax of the predefined XVC manager test scenario description language used to implement XVC tests. Table C-1. summarizes the convention used to describe the syntax of the various commands. All other textual elements as specified as-is.

File Structure

A file is composed of commands, comments and blank lines, each terminated by the newline or end-of-file character. A command may span multiple lines if the newline is escaped using a backslash (\) character.

Table C-1. Grammar Notation

Simulation Handling	Action
TOKEN	Case-insensitive, predefined token
<token>	Required user-defined token
[token]	Optional token
{token}	Optional token that can be specified 0 or more times
(tokens|tokens)	A mandatory choice of tokens

Lines are composed of tokens. Tokens are blank-separated strings. Tokens may contain environment variable substitutions. Blanks may be included in a token by quoting it using double quotes. A quoted string may contain a double-quote character by escaping it using a backslash.

The commands *VERBOSITY*, *STOPONERROR* and *STOPONEVENT* must appear, in any order or number, before any other commands.

The commands *ACTION* and *INTERRUPT* must appear after a *SCENARIO* command.

The *EXECUTE* commands must be the last commands.

All other commands can appear anywhere in the command sequence.

User-defined tokens, such as XVC names or action descriptions, are case-sensitive. All predefined tokens, such as *DISPLAY* or *WAIT*, are case insensitive.

Example C-1. Typical Command Sequence

```
VERBOSITY ...
STOPONERROR ...

LOG ...

MAPEVENT ... GLOBAL ...
EVENT ... GLOBAL ...

SCENARIO ...
     EVENT ...
     ACTION ...
     ACTION ...
```

```
        INTERRUPT ...

    SCENARIO ...
        EVENT ...
        MAPEVENT ...
        ACTION

    EXECUTE ...
```

Environment Variables

Any token can contain environment variable substitution.

Syntax:

```
... ${<env_name>}
```

where *<env_name>* is the name of an environment variable. It is an error if the variable is not set.

The content of the environment variable is included in the token. An environment variable containing blank characters is not interpreted as multiple tokens. A dollar sign ($) can be specified by escaping it using a double dollar sign ($$).

Commands

Comments

Specifies arbitrary text that is ignored by the XVC manager.

Syntax:

```
// {<string>}
```

A comment is terminated by the next newline character. Any character following the "//" is ignored. Commands may be specified before the "//" characters.

Example:

```
// This is a comment line, followed by a blank line

ACTION xvc a "write 0x400 0x55AA" // Comment on action
```

#include

Includes a scenario definition file in the current scenario definition file.

Syntax:

```
#INCLUDE <filename>
```

where <*filename*> is the name of a file that will be included as if its entire content had been specified instead of the *#include* command. Included files may include other files. If the filename is a relative path, the path is interpreted as relative to the location of the file containing the *#include* directive, not necessarily the current working directory.

It is illegal for an included file to contain *X*, *QUITON* or *LOG* commands or global notification definitions.

#define

Defines a symbolic name for a numeric identifier to aid the readability and maintainability of test command files.

Syntax:

```
#define <symbol> (<nid>|<sid>|<sev>|<sid>.<sev>|<gev>)
```

where <*symbol*> is an alphanumeric string with no spaces or control characters.

<*nid*>, <*sid*>, <*sev*> and <*gev*> are unsigned integer values referring to a local XVC notification identifier, scenario identifier, scenario event or global event, respectively.

VERBOSITY

Sets the global message verbosity level.

Syntax:

```
VERBOSITY (<instance>|ALL) <severity_level>
```

where <*instance*> is a string or regular expression specifying the instance names of the XVCs from which messages of the specified severity or higher should be displayed. Specifying *ALL* is identical to specifying */./*.

<severity_level> specifies the minimum severity levels of messages to be displayed. Messages of a lower severity level will not be displayed. For further information, see the *vmm_log::set_verbosity()* method on page 374.

LOG

Writes messages issued from specified XVC instances into a user-specified file. This includes messages issued through the trace message service interfaces.

Syntax:

```
LOG (<instance>|ALL) <filename>
```

where *<instance>* is a string or regular expression specifying the instance names of the XVC from which messages should be logged to the specified file. Specifying *ALL* is identical to specifying */./*. A single XVC instance can log messages to more than one file.

<filename> is the name of the file to which the messages will be written. If the name is prefixed with a '+' character, the messages are appended to the file. If the name is prefixed with a '-', the messages are no longer written to the specified file.

Examples:

```
LOG ALL "Messages.log"
LOG UART "+UART.log"
...
LOG UART "-UART.log"
```

LOG NONE

Stops writing messages to files and closes all log files. Applies to all XVC instances and all files. This action includes messages issued through the trace message service interfaces.

Syntax:

```
LOG NONE
```

XVCTRACE

Writes messages issued from specified XVC instances via their *xvc_xactor::trace* message service interface or the XVC manager via its *xvc_manager::trace* message service interface into a user-specified file.

Syntax:

```
XVCTRACE (<instance>|MANAGER|ALL) <filename>

XVCTRACE NONE
```

where *<instance>* is a string or regular expression specifying the instance names of the XVC from which trace messages should be logged to the specified file. Specifying *ALL* is identical to specifying */./*. *MANAGER* specifies the XVC manager itself. If *NONE* is specified, trace messages are no longer logged to a file.

COVFILE

Specifies the name of the functional coverage database file.

Syntax:

```
COVFILE (<dbname>|NONE)
```

where *<filename>* is the name of the database to which coverage information will be written. If *NONE* is specified, no functional coverage is to be collected during simulation.

STOPONERROR

Stops the simulation after *<count>* error messages have been issued. By default, the simulation stops after issuing 10 messages with a severity of *ERROR_SEV*. The simulation always stops when a message with a *FATAL_SEV* severity level is issued.

Syntax:

```
STOPONERROR <count>
```

STOPONEVENT

Stops the simulation if the specified circumstance occurs before the natural end of the test.

Syntax:

```
STOPONEVENT [<sid>.]<sev> \
   <(IMMEDIATE|GRACEFUL)> [<count>]
```

where *[<sid>.]<sev>* specifies a a global or local event.

If *IMMEDIATE* is specified, the simulation is stopped as soon as the specified event is indicated. If *GRACEFUL* is specified, the XVC manager will delay the end of the simulation until all actions executing in the current scenario have completed.

<count> is the number of times the specified events must be indicated before the simulation is stopped.

SCENARIO

Defines a new test scenario and implicitly terminates the definition of a previous test scenario.

Syntax:

```
S[CENARIO] <sid> [<description>]
```

where *<sid>* is a unique unsigned integer identification number for the scenario to be defined. Two scenarios cannot have the same identification number in the same simulation.

<description> is an arbitrary description string that will be displayed to the simulation output and log file(s) whenever that scenario is executed.

This command starts the definition of a new scenario. Any subsequent *EVENT*, *MAPEVENT*, *ACTION* or *INTERRUPT* commands will define this scenario. A subsequent *SCENARIO* or *EXECUTE* command terminates the scenario.

Actions may not execute in the sequence they are specified in the scenario. Actions targeted to different XVCs will execute concurrently, as soon as the target XVC can execute the next action. Only actions targeted to the same XVC will execute in the sequence specified in the scenario.

It is an error to define a scenario that does not contain at least one action.

EVENT

Defines a local or global event. It allows actions of different XVCs to be coordinated within a scenario. Only global events can be defined outside of a scenario.

Syntax:

```
E[VENT] [ONESHOT] [(LOCAL|GLOBAL)] (<sev>|<gev>) IS
   [<sid>.]<sev>{(,|+)[sid.]<sev>} [<descr>]
```

where *<sev>* or *<gev>* is a unique unsigned identification number of the local or global event to be defined. Global events must have globally-unique identifiers. Scenario events must have scenario-unique identifiers. A scenario event with the same identifier as a global event will hide the global event within that scenario.

If *ONESHOT* is specified, the event will be indicated only once during the entire test, even though the criteria for notification may occur multiple times. By default, the event is notified each time the criteria occurs.

If *LOCAL* is specified, the event is local to the current scenario. If *GLOBAL* is specified, the event is global to all scenarios. By default, events are local. Global events can be defined either outside or inside a scenario.

The event is notified when the specified notifications have been observed to be notified. *<sev>{+<sev>}* specifies that the defined event is indicated when all of the specified events are indicated, in any order. *<sev>{,<sev>}* specifies that the defined event is indicated when any of the specified events are indicated. The + operator has precedence over the , operator. When defining global events, events local to scenarios can be referred to by prefixing them with the appropriate scenario identifier using the *<sid>.* notation.

<descr> is an optional string message, which will be displayed in a *DEBUG_TYP*, *TRACE_SEV* trace message if and when the notification is indicated. If no description is specified, a default description is used.

Example:

```
EVENT 3 IS 2 + 1
```

Declares local scenario event 3 that will be indicated following the occurrence of the events 2 and 1.

```
E GLOBAL 6 IS 2,3.1 "Setup complete"
```

Declares a global event 6 that will be indicated, with the message “Setup complete” displayed, as soon as either global event 2 or the local event 1 in scenario 3 have been indicated.

MAPEVENT

Maps a local XVC event to scenario or global event. It allows the actions of different XVCs to be coordinated within a scenario.

Syntax:

```
M[APEVENT] [ONESHOT] [(LOCAL|GLOBAL)] (<sev>|<gev>) IS \
   <xid> E[VENT] <nid> [<descr>]
```

where *<sev>* or *<gev>* is a unique unsigned identification number of the local or global event to be defined. Global events must have globally-unique identifiers. Scenario events must have scenario-unique identifiers. A scenario event with the same identifier as a global event will hide the global event within that scenario.

If *ONESHOT* is specified, the event will be indicated only once during the entire test, even though the criteria for notification may occur multiple times. By default, the event is notified each time the criteria occurs.

If *LOCAL* is specified, the event is local to the current scenario. If *GLOBAL* is specified, the event is global to all scenarios. By default, events are local. Global events can be defined either outside or inside a scenario.

<xid> is a string or regular expression specifying the instance name of the XVCs that is the source or cause of this event.

<nid> identifies the local event using the notification identifier in the *xvc_xactor::notify* notification service interface that, when indicated, will indicate this event.

<descr> is an optional string message, which will be displayed in a *DEBUG_TYP*, *TRACE_SEV* trace message if and when the event is indicated. If no description is specified, a default description is used.

Example:

```
Mapevent 3 is "CLCD" event 1
```

Defines a local event 3 that will be indicated whenever the XVC instance named “CLCD” indicates its local notification 1.

```
MAPEVENT GLOBAL 5 is /^AHB/ event 5 "Abort"
```

Defines a global event 5 that will be indicated whenever any one of the XVCs with an instance name matching the regular expression indicates its local notification 5.

ACTION

Adds an action execution to the scenario definition.

Syntax:

```
A[CTION] <instance> <action> \
   [W[AIT] <wait_for>] [E[MIT] <notification>]
```

where *<instance>* is the instance name of the XVC for which the action is intended. Regular expressions cannot be used because actions cannot be targeted to multiple XVC instances.

<action> is the string that defines the action to be executed. The syntax of this token is action-specific and is defined in the *xvc_action::parse()* method of the relevant action descriptor.

The <wait_for> and *<notification>* tokens are described in the *WAIT* and *EMIT* options sections.

This command adds an action to the action list of the specified XVC instance. The *WAIT* option lets the start of the action be synchronized with the indication of one or more specified scenarios or global events(s). The *EVENT* options lets the completion of that action indicate the specified local notification.

Examples:

```
ACTION ARM_DMA "5 True enable 5 9" WAIT 1
```

The XVC instance named *ARM_DMA* waits for the next indication of the scenario or global event ‘1’, then it will execute the specified action.

```
ACTION uart "enable master mode" EMIT 4
```

The XVC instance named *uart* will execute the specified action. When the action completes, the local XVC notification ‘4’ will be indicated.

INTERRUPT

Add a high-priority action execution to the scenario definition.

Syntax:

```
I[NTERRUPT] [ONESHOT] <instance> <action> \
   [W[AIT] <wait_for>] [E[MIT] <notification>]
```

where *<instance>* is the instance name of the XVC for which the interrupt action is intended. Regular expressions cannot be used because actions cannot be targeted to multiple XVC instances.

<action> is the string that defines the interrupt action to be executed. The syntax of this token is action-specific and is defined in the *xvc_action::parse()* method of the relevant action descriptor.

If *ONESHOT* is specified, the interrupt action will only execute one on the first occurrence of the specified event. Subsequent occurrences of the event will not cause the interrupt action to re-execute

The <wait_for> and *<notification>* tokens are described in the *WAIT* and *EMIT* options sections.

Schedule the specified action for execution whenever the specified *WAIT* condition is observed. If no WAIT condition is specified, the interrupt action is scheduled once immediately. The interrupt action will interrupt the execution of any action that is currently executing by the target XVC instance. The granularity of the interruption is defined by the implementation of the to-be-interrupted action in the *xvc_action::execute()* method by invoking the *xvc_xactor::wait_if_interrupted()* method. Interrupt actions cannot be interrupted.

If two interrupt actions are scheduled to execute on the same XVC instance at the same time, they will be executed in a non-deterministic order.

Examples:

```
INTERRUPT ARM_DMA "Reconfigure Generation" W 1
```

The XVC instance named *ARM_DMA* will execute the specified action whenever global or scenario event ‘1’ is indicated, at the earliest possible action interruption point.

```
INTERRUPT ARM_IAHB "Reset ~/nIRQ"  E 4
```

The XVC instance named *ARM_DMA* will immediately execute the specified interrupt action at the next interruption opportunity. When the action completes, the local XVC notification ‘4’ will be indicated.

... WAIT <wait_for>

This is an option for the *ACTION* and *INTERRUPT* command, not a stand-alone command. It is used to specify a global or scenario event indication for which the XVC will wait before starting the action.

Syntax:

```
... W[AIT] <sev>|<gev>
```

where *<sev>* or *<gev>* is the identifier for a scenario or global event.

The *ACTION* or *INTERRUPT* command will be delayed until the specified event has been indicated.

Example:

```
ACTION ... W 1
```

... EMIT <indication>

This is an option for the *ACTION* and *INTERRUPT* command, not a stand-alone command. It is used to specify a *xvc_xactor::notify* notification that is indicated when the XVC completes the execution of an action.

Syntax:

```
... E[MIT] <nid>
```

where *<nid>* is the identifier for a notification in the *xvc_xactor::notify* notification service interface of the XVC executing the action.

Example:

```
ACTION ... EVENT 1
```

When the execution of the action completes, notifications ‘1’ will be indicated, potentially allowing other XVC actions to be executed if they are mapped to global or scenario events.

EXECUTE

Executes the specified scenarios, in sequence.

Syntax:

```
[E]X[ECUTE] <sid>{ <sid>}
```

where <*sid*> is the unique identifier of a scenario to execute.

Example C-2. Example Test Scenario Description File

```
// Test Set-up
//-----------------------------------------------------
VERBOSITY ALL TRACE
LOG ALL "Trace.txt"
COVFILE NONE
STOPONERROR 1

// Global events
MAPEVENT GLOBAL 1 IS a EVENT 1

//Scenario definitions
//-----------------------------------------------------
scenario 1 "Demonstrating XVCs"


action a "Action 1"
action b "Action 1"


scenario 2 "My second scenario - action end events"

action a "Action 2" E 1
action b "Action 2" W 1


scenario 3 "My third scenario - interrupt actions"

action xvc a "Action 1" E 1
interrupt xvc b "Action 1" W 1


// Execution
//-----------------------------------------------------
x 1 2
x 3
```

APPENDIX D SOFTWARE TEST FRAMEWORK

This appendix specifies the detailed behavior of a C library that can be used to implement a VMM-compliant software verification environment and tests. The actual implementation of these classes is left to each tool or platform provider.

BASIC TYPES

The following are definitions of standard types, with explicit bit widths for clarity:

```
typedef unsigned int        BOOL;
typedef signed   char       BYTE8;
typedef unsigned char       UBYTE8;
typedef signed   short      HWD16;
typedef unsigned short      UHWD16;
typedef signed   int        WORD32;
typedef unsigned int        UWORD32;
typedef signed   long long  LLONG64;
typedef unsigned  long long   ULLONG64;
```

A *word* is defined to be 32-bits long.

SYSTEM DESCRIPTOR

The system descriptor is an array of peripheral device descriptors.

```
const svSYS_SystemElement svSYS_SystemDescriptor[] =
{...}
```

A symbol must exist that specifies the index of a peripheral descriptor in the system descriptor. The symbol must be named *svSYS_Element_XXXX_N* where *XXXX* is the name of the peripheral and *N* is the instance number.

Peripheral Descriptor

A *svSYS_SystemElement* is a descriptor for an instance of a peripheral device. Each peripheral instance is described using the following structure:

```
typedef const struct SystemElement svSYS_SystemElement;
struct SystemElement {
   UWORD32                  Tested;
   svSYS_SystemID           DeviceID;
   UWORD32                  BaseAddress;
   svSYS_SystemInterrupts   Interrupts;
   svSYS_SystemClocks       Clocks;
   svSYS_SystemDMA          DMA;
   svSYS_ActionSheetItem   * pActionSheet;
   svSYS_CheckState         pCheckState;
   UWORD32                  Padding[x];
};
```

The following is an example of a fully specified peripheral descriptor in a system descriptor:

```
const svSYS_SystemElement svSYS_SystemDescriptor[] = {
   ...
   //Data for P123 UART instance #0
   {
      0x1,                    // Selected for testing
      "P123",                 // Device ID "P123"
      0x20000000,             // Base Address
      {{svSYS_GET_SYS_DATA(P123, 0), 12}},
                              // One interrupt at
                              // controller 0, source 12
      {4000000},              // One clock at 4MHz
      {{&svSYS_GET_SYS_DATA(P456, 0), 1, 4}},
                              // One DMA at
                              // controller 0, channel 4
                              // accessible via port 0
      ActionSheet_P456,       // Address of action sheet
      svP456_CheckState       // State checking function
    }
    ...
}

typedef enum {
   ...
   svSYS_Element_P123_0,
   ...
   svSYS_Element_P456_0,
   ...
}
```

Tested

This structure property identifies peripheral sets this peripheral belongs to. A peripheral belongs to peripheral set *N* if bit *N* of this structure property is set. A peripheral can thus belong to more than one peripheral set.

DeviceID

This structure property specifies the name of the peripheral, for example “P123”. It is a null-terminated string padded with zeros if it is less than seven characters.

```
typedef unsigned char svSYS_SystemID[8];
```

BaseAddress

This structure property specifies the hardware base address of the peripheral.

Interrupts

This structure property specifies the interrupts generated by the peripheral. A peripheral can generate interrupts to up to eight different interrupt controllers.

```
typedef svSYS_SystemInterrupts
svSYS_SystemInterrupts[8];
```

Each entry in the array describes a single interrupt connection from the device to a controller. See “Interrupt Descriptor” on page 463 for a description of the interrupt descriptor.

Clocks

This structure defines the frequency of the clocks signals supplied to the peripheral. Each element of the array is a clock rate in Hz. If a clock signal is not used, the frequency value is specified as 0.

```
typedef UWORD32 svSYS_SystemClocks[4];
```

DMA

This structure property describes up to four DMA channels connected to the peripheral.

```
typedef SYS_SystemDMA svSYS_SystemDMA[4];
```

Each entry in the array describes a single DMA connection from the device to a controller. See “DMA Channel Descriptor” on page 464 for a description of the DMA channel descriptor.

pActionSheet

This structure property is a pointer to zero-terminated array of actions called an *action sheet*. These actions—and only these actions—are available for execution on this peripheral. Multiple instances of the same peripheral may refer to the same action sheet.

```
const svSYS_ActionSheetItem ActionSheet_P123 [] = {
   {&MyP123FirstAction, &MyP123FirstAction_Level,
    &MyP123FirstAction_Name},
   {&MyP123SecondAction, &MyP123SecondAction_Level,
    &MyP123SecondAction_Name},
   ...
   {0} //Must be zero terminated
}
```

See “svSYS_ActionSheetItem” on page 467 for a specification of the action sheet entry.

pCheckState

This structure property is a pointer to a function that checks if the peripheral is in an idle state.

```
typedef svSYS_eTestResponse(* svSYS_CheckState)
   (svSYS_SystemElement * pPeriph)
```

The function returns *svTestPassed* if the peripheral is currently in an idle state and *svTestFailed* otherwise. The meaning of *idle* is specific to each peripheral but in general, *idle* will mean that the peripheral is disabled and has no interrupts flagged.

See “svSYS_eTestResponse” on page 466 for the specification of *svSYS_eTestResponse*.

Padding

This structure property pads the value of the structure to a power-of-two number of words if needed. It will allow rapid iteration through the array of peripheral descriptors.

Interrupt Descriptor

This structure describes a single interrupt connection from a peripheral to an interrupt controller. Peripheral interrupts are described by the *Interrupts* struct property of that peripheral's descriptor, as specified in “Peripheral Descriptor” on page 460.

```
typedef struct {
   svSYS_SystemElement  * pController;
   UWORD32                Source;
} svSYS_SystemInterrupts;
```

pController

This structure property is a pointer to the peripheral descriptor for the interrupt controller. If the value is *NULL*, there is no interrupt described and this descriptor is to be ignored.

Source

The interrupt number, in the interrupt controller, of the interrupt that is generated by the peripheral.

DMA Channel Descriptor

This structure describes a single DMA channel connection between a peripheral and a DMA controller. The DMA channels for a peripherals are specified by the DMA struct property in that peripheral's descriptor. See “Peripheral Descriptor” on page 460 for more details.

```
typedef struct {
   svSYS_SystemElement  * pController;
   UHWD16                 Masters;
   UHWD16                 Channel;
} svSYS_SystemDMA;
```

pController

This structure property is a pointer to the peripheral descriptor for the DMA controller. If the value is *NULL*, there is no DMA channel described and the descriptor must be ignored.

Masters

This structure property indicates which of the master ports of the DMA controller can access the DMA channel. Master ports are identified by their corresponding bit position. For example, a value of 0x5 specified that the DMA channel to the peripheral is accessible from master ports 0 and 2.

Channel

The DMA channel number, in the DMA controller, of the DMA channel to the peripheral.

TEST ACTIONS

The following macros, declarations and functions are available to support the specification and execution of test actions on specific peripherals.

svTEST_NAME()

This macro is used to specify an arbitrary string to describe a test action. The string is accessible as an externally visible character array named *<actionname>_Name*. It is used to identify the running test action in simulation messages and the debug channel.

```
#define svTEST_NAME(action, descr) const char
action##_Name[] = descr;
```

The *action* parameter is the name of the function implementing the test action. The *descr* parameter is a string literal which describes or identifier the test action. For example:

```
svTEST_NAME(svP123_FirstAction,
            "Test Interrupts on P123")
```

svTEST_LEVEL()

This macro is used to specify a complexity-level value to a test action. The value is accessible as an externally visible unsigned integer named *<actionname>_Level*.

```
#define svTEST_LEVEL(action, level) \
   const unsigned int action##_Level = level;
```

The *action* parameter is the name of the function implementing the test action. The *level* parameter is an integer value specifying the complexity level of the test action. For example:

```
svTEST_LEVEL(svP123_FirstAction, 3)
```

svSYS_GET_SYS_DATA()

This macro returns the address of the peripheral descriptor corresponding to the specified instance of the specified peripheral. The macro uses the specified instance and peripheral name to construct the symbolic name of the index of the peripheral descriptor in the system descriptor.

```
#define svSYS_GET_SYS_DATA(Device, DeviceNum) \
   (svSYS_SystemDescriptor +
    svSYS_Element_##Device##_##DeviceNum)
```

svSYS_eTestResponse

The *svSYS_eTestResponse* type is used to specify the result for a test action, as described in Table D-1.

```
typedef enum {
   svSYS_TestSkipped    = -1,
   svSYS_TestFailed     = 0,
   svSYS_TestPassed     = 1,
   svSYS_TestException  = 2,
   svSYS_TestAvailable  = 3
} svSYS_eTestResponse;
```

Table D-1 Test Status Codes

svSYS_TestSkipped	Test not performed
svSYS_TestFailed	Test failed
svSYS_TestPassed	Test passed
svSYS_TestException	Test generated an exception
svSYS_TestAvailable	Test marked as available

svSYS_SeqTest

This pointer-to-function type is a prototype for test action functions.

```
typedef svSYS_eTestResponse (*svSYS_SeqTest)
   (svSYS_SystemElement * pPeriph);
```

The *pPeriph* parameter is a pointer to the system descriptor for the peripheral targeted by this test action. The action is responsible for retrieving and using the fields of interest.

svSYS_ActionRun()

Runs the specified test action on the specified peripheral.

```
void SYS_ActionRun (
   char                 * pTestName,
   char                 * pDeviceName,
   svSYS_SeqTest          Action,
   svSYS_SystemElement  * pPeriph
};
```

The *pTestName* parameter is a pointer to the description or name of the test action to be executed. The *pDeviceName* parameter is a pointer to the descriptor or name

of the target peripheral. The *Action* parameter is a pointer to the test action function to be executed. The *pPeriph* parameter is a pointer to the peripheral descriptor of the target peripheral.

This function standardizes the behavior and output of actions and performs common pre- and post-action operations:

- Outputs a debug message containing *pTestName*, *pDeviceName* and *pPeriph->BaseAddress*
- Configures the exception handler to use a test-specific routine
- Runs the *pPeriph->pCheckState()* function to check that the peripheral is in an idle state and outputs a debug message describing the result
- Calls the test action, passing in the *pPeriph* pointer
- Restores the original exception handler
- Outputs a debug message describing the test result
- Adds a new entry to the test summary report

svSYS_ActionSheetItem

This structure defines a test action that is part of a test action sheet. A test action sheet is specified as a zero-terminated array of *svSYS_ActionSheetItem*. See “pActionSheet” on page 462.

```
typedef struct {
   svSYS_SeqTest   TestFunction;
   WORD32        * pLevel;
   char          * pName;
} svSYS_ActionSheetItem;
```

The *TestFunction* structure property is a pointer to the function implementing the test action. The *pLevel* structure property is a pointer to the test complexity-level value for the test action. The *pName* structure property is a pointer to the description or name of the test action.

svSYS_ActionSheetRun()

This function is used to execute the test actions found in a specific test action sheet.

```
void SYS_ActionSheetRun (
   char                * pDeviceName,
   UWORD32               Sequence,
   svSYS_eTestOrder      TestOrder,
   svSYS_SystemElement * pPeriph
);
```

The *pDeviceName* parameter is a pointer to the descriptor or name of the target peripheral. The *Sequence* parameter specifies the complexity levels that are to be executed. The *testOrder* parameter specifies the order in which to execute the test actions found in the action sheet. The *pPeriph* parameter is a pointer to the peripheral descriptor of the target peripheral.

This function will iterate through all test actions found in the action sheet *pPeriph->ActionSheet* in the order specified by the *TestOrder* parameter. If the bit in the *Sequence* value corresponding to the complexity level of the test is set, the test is executed. Otherwise, the test is skipped and a new entry is added to the test summary report.

svSYS_eTestOrder

This enumerated type is used to specify the order of execution of tests in an action sheet.

```
typedef enum {
   svSYS_TestSequence,
   svSYS_TestReverse,
   svSYS_TestRandom
} svSYS_eTestOrder;
```

The execution order specified by each value is described in the following table:

Table D-2 Test Sequencing Codes

`svSYS_TestSequence`	Execute tests in order specified in the action sheet
`svSYS_TestReverse`	Execute tests in the reverse order specified in the action sheet
`svSYS_TestRandom`	Execute tests in random order

`svSYS_AllActionSheetsRun()`

This function iterates through all peripherals in the system descriptor in a specified order and executes all tests in the test action sheet for each peripheral.

```
void svSYS_AllActionSheetsRun (
   UWORD32          Peripherals,
   UWORD32          Sequence,
   svSYS_eTestOrder PeriphTestOrder,
   svSYS_eTestOrder ActionTestOrder);
```

The *`Peripherals`* parameter is specifies which peripherals should be tested in the run. The *`Sequence`* parameter specifies the complexity levels that are to be executed. The *`PeriphTestOrder`* parameter specifies the order in which to test the peripherals found in the system descriptor. The *`ActionTestOrder`* parameter specifies the order in which to execute the test actions found in the action sheet.

Peripherals are selected by and-ing the value of the peripherals parameter to the peripheral's tested value. If the result is non-zero, the action sheet associated with the peripheral is run. If the result is zero, the peripheral is skipped. This mechanism creates 32 different sets of peripherals that can be included or excluded from a test run. If a peripheral is selected, its action is sheet is run.

`UWORD32 svSYS_Peripherals;`
`UWORD32 svSYS_Sequence;`

These global variables may be polled by the software test framework after every execution of *`svSYS_AllActionSheetsRun()`*. If the value of either variable has changed since the last poll, the *`svSYS_AllActionSheetsRun()`* is invoked again. If either is zero or the values have not have changed, the polling continues until a change is detected.

These variables lets different peripherals or tests be dynamically selected using a debugger. If the execution of the software test is stopped and new values inserted, a new set of tests will be run after the completion of the current test run.

LOW-LEVEL SERVICES

svSYS_DebugLevel

Global variable used to specify the current debug level for the software test. A value of *svSYS_FATAL_SEV* is invalid and interpreted as *svSYS_ERROR_SEV*. The value of this variable can be modified at run time or via a debugger.

```
svSYS_eMsgLevel svSYS_DebugLevel = svSYS_NORMAL_SEV;

typedef enum {
   svSYS_FATAL_SEV,
   svSYS_ERROR_SEV,
   svSYS_WARNING_SEV,
   svSYS_NORMAL_SEV,
   svSYS_TRACE_SEV,
   svSYS_DEBUG_SEV,
   svSYS_VERBOSE_SEV
} svSYS_eMsgLevel;
```

svSYS_Printf()

This function is used to issue all messages from a software test.

```
void svSYS_Printf(svSYS_eMsgLevel     Level,
                  svSYS_ePrintType    Type,
                   void              * pParam);
```

This function will print the specified message to the defined output channel if the *Level* parameter is less than or equal to the current verbosity level defined by the *svSYS_DebugLevel* variable.

The *Type* parameter specifies the format of the data pointed to by the *pParam* parameter, using one of the following enumerated values:

```
typedef enum {
   svSYS_PrintString,
   svSYS_PrintHex,
   svSYS_PrintDec,
   svSYS_PrintBoolean
} svSYS_ePrintType;
```

svSYS_EnvironmentGet()

This function identifies the software verification environment used to execute the software tests.

```
svSYS_eEnvironment svSYS_EnvironmentGet(void);
```

The value returned by this function is one of the following enumerated values:

Table D-3 Verification Environment Identification Codes

svSYS_EnvSimulator	Simulation-based environment
svSYS_EnvFPGA	FPGA-based environment

svSYS_Ignore()

This function verifies if the current verification environment matches one of the specified unsupported verification environments. It is used to skip tests that cannot be executed on specific verification environments.

```
BOOL svSYS_Ignore(UWORD32 TestMask);
```

The *TestMask* parameter is the bitwise-OR of the enumerated values corresponding to the unsupported environments. If the current environment matches one of the specified unsupported environments, a *svSYS_NORMAL_SEV* "Test Skipped" message is issued and *TRUE* is returned. Otherwise, *FALSE* is return. The test action code is responsible for aborting the test if it is to be ignored.

```
if (svSYS_Ignore(svSYS_EnvFPGA | svSYS_EnvPV)) return;
```

svSYS_HARDCODED()

This macro has no effect on the code but identifies potential porting issues.

```
#define svSYS_HARDCODED(code) code
```

svSYS_Malloc()
svSYS_Free()

The software framework does not implement a full heap, due to the overheads associated with managing system resources. Instead, it offers the following functions:

```
void * svSYS_Malloc(unsigned int size);
void svSYS_Free(void *);
```

There must be a block of data for dynamic memory allocation, which is identified in a linker definition file as execution region heap. The heap would be located from a heap base address constant to a heap limit-1.

Where memory leakage is not an issue, a simple heap management process can be used. The heap manager maintains a pointer to the top of the current heap. When *svSYS_Malloc()* is called, the heap is checked for the requested space, then the current pointer is returned and the pointer is incremented by the requested space size. Freed dynamic memory is never reclaimed and the *svSYS_Free()* function may be implemented as a blank macro.

In platforms and environments where memory leak is an issue, these functions may implement a more complex dynamic memory management system, including garbage collection.

svIO_BYTE_READ()
svIO_BYTE_WRITE()
svIO_WORD_READ()
svIO_WORD_WRITE()

The software framework provides macros for accessing memory-mapped registers, given the base address and an offset in bytes.

For word-width (32-bit) registers, the following macros are available:

```
typedef volatile UWORD32 svRegister;

#define svIO_WORD_READ(_Addr, _Offset)\
          *((svRegister*) ((UWORD32)_Addr +
                           (UWORD32)_Offset))

#define svIO_WORD_WRITE(_Addr, _Offset, _Value) \
          *((svRegister*) ((UWORD32)_Addr +
                           (UWORD32)_Offset)) = \
                               (UWORD32)_Value;
```

For byte-width registers the following are available:

```
typedef volatile UBYTE8 svRegisterByteAccess;

#define svIO_BYTE_READ(_Addr, _Offset) \
       *((svRegisterByteAccess*) ((UWORD32)_Addr +
                                  (UWORD32)_Offset))

#define svIO_BYTE_WRITE(_Addr, _Offset, _Value) \
        if(_Value != (_Value & 0xFF)) SYS_swi();\
        *((svRegisterByteAccess*) \
           ((UWORD32)_Addr + (UWORD32)_Offset)) =
              (UBYTE8)_Value;
```

svSYS_RAND()

This function will generate and return a pseudo-random number. It is used by the *svSYS_AllActionSheetsRun()* function for sequencing tests in random order. It is also available for use within test actions to create random tests.

```
UWORD32 svSYS_RAND(UWORD32 Limit)
```

The returned value will be between 0 and *Limit*-1. It is recommended that *Limit* be a power of 2 for runtime efficiency. The implementation of this function may invoke the standard C *rand()* function. The seed is set during initialization by the bootstrap module.

svSYS_BlockCopy()

This function copies a block of 32-bit words from one location to another.

```
void svSYS_BlockCopy(UWORD32  * pSourceAddr,
                     UWORD32  * pDestAddr,
                      UWORD32    NumWords);
```

This function may be implemented using processor instructions or DMA channels.

svSYS_ThrowException()

This function is used to throw a software interrupt (SWI) exception.

```
svSYS_ThrowException()
```

This function hides the system-specific way of generating a processor exception and catches this exception.

svSYS_ASSERT()

This macro is used to verify that a condition holds true and throws an *SWI* exception otherwise.

```
svSYS_ASSERT(BOOL Assertion)
```

If the value of the *Assertion* parameter is *FALSE*, an *SWI* exception is thrown.

Cache Lockdown

Some processors have the ability to lock down and free either or both the instruction and data cache. This ability ensures that there will be no instruction or data bus activity while a cache is locked down.

Cache lockdown is implemented using the following functions:

svSYS_CacheLock()

This function locks all addresses between the start and end addresses into cache. The start address is included. The end address is normally excluded but may be locked down if cache line size requires it.

```
typedef UWORD32 svSYS_CacheLines;
typedef enum {
   svSYS_InstructionCache,
   svSYS_DataCache
} svCacheType;
svSYS_CacheLines svSYS_CacheLock(
   svSYS_CacheType    eCache,
   void             * pStartAddress,
   void              * pEndAddress);
```

The return value that describes the block of locked cache lines is specified Table D-4.

If there is insufficient cache space to lock the requested address range, the maximum available number of cache lines should be locked, leaving at least one free cache line. It is recommended that a debug message will be generated to indicate this occurrence.

It is assumed that cache locking and unlocking will remain the responsibility of the user.

Table D-4 Cache Lines Descriptor

`Bits 7-0`	First cache line number
`Bits 15-8`	Last cache line number
`Bits 23-16`	Unused
`Bits 31-24`	0: Instruction cache 1: Data cache

svSYS_CacheUnlock()

This function unlocks a set of cache lines.

```
void svSYS_CacheUnlock(svCacheLines LockedLines);
```

The descriptor for the cache lines to unlock, specified by the *LockedLines* parameter, is identical to the descriptor returned by the *svSYS_CacheLock()* function.

svSYS_CACHE_BLOCK_START()
svSYS_CACHE_BLOCK_END()

Macros used to name a block of instructions that can be locked in the instruction cache.

```
#define svSYS_CACHE_BLOCK_END(name) \
   void name##_CacheBlockEnd(void) {}

#define svSYS_CACHE_BLOCK_END(name) \
   void name##_CacheBlockEnd(void) {}
```

These macros create symbols that can be used to specify start and end addresses to the *svSYS_CacheLock()* function.

Interrupt Controller

Software test actions can configure interrupt controllers to verify the interrupt sourcing features of a peripheral. Note that this is not done to test the interrupt controller, where controller-specific actions will be needed, but to test a peripheral generating an interrupt to the interrupt controller.

The normal process of using interrupts is to bind an interrupt handler to an interrupt. An interrupt handler is a software routine that is executed when the interrupt occurs.

The following example illustrates how to use some of the interrupt support routines to enable interrupts before executing the test and disables them afterwards:

```
void MyTestHandler(UWORD32                Source,
                   svSYS_SystemElement *  pPeriph)
{
   ... // Interrupt handler during test action */
}

svSYS_eTestResponse
P123_InterruptTest(svSYS_SystemElement * pPeriph)
{
   svSYS_InterruptAllBind(&MyTestHandler, pPeriph);

   ... // Test action using interrupts

   svSYS_InterruptAllUnBind(&MyTestHandler, pPeriph);
}
```

The following functions are available to configure interrupt controllers:

svSYS_InterruptInit()

Initializes the specified interrupt controller.

```
void svSYS_InterruptInit(
   svSYS_SystemElement  * pController,
   BOOL                   Enabled);
```

The *pController* parameter specifies the interrupt controller to be initialized. This function must be invoked before tests that use the interrupt controller.

If the *Enabled* parameter is *FALSE*, all interrupts will be disabled at the interrupt handler. If *TRUE*, all interrupts not already asserted will be enabled. After initialization, the handler for all interrupt sources is set to the following routine:

```
void SYS_BadInterrupt(UWORD32 Source);
```

This routine will generate an exception, indicating that an unexpected interrupt has occurred.

svSYS_InterruptHandler

The interrupt handler routine must comply with the following prototype:

```
typedef __irq void (*svSYS_InterruptHandler)(
   UWORD32               Source,
   svSYS_SystemElement  *  pPeriph);
```

The *Source* parameter is the number of the interrupt, in the controller, that is being handled. The *pPeriph* parameter specifies the peripheral descriptor for the peripheral generating the interrupt.

svSYS_InterruptBind()

This function binds a handler to an individual interrupt and enables the interrupt.

```
void svSYS_InterruptBind(
   svSYS_InterruptHandler   Handler,
   svSYS_SystemElement    * pPeriph,
   UWORD32                  IntIndex,
   svSYS_eInterruptType     IntType,
   UBYTE8                    priority);
```

The *Handler* parameter specifies the interrupt handler routine to be run when the interrupt occurs. The *pPeriph* parameter specifies the peripheral descriptor of the peripheral generating the interrupt. The *IntIndex* parameter specifies the interrupt number in the peripheral's list of interrupts. The *IntType* parameter specifies the type of interrupt (either *svSYS_IRQ* or *svSYS_FIQ*). And the *Priority* parameter specifies the priority level of the interrupt on a scale 0 (low) to 15 (high).

svSYS_InterruptUnBind()

This function unbinds an interrupt handler routine from an individual interrupt, restoring the unexpected interrupt handler and leaving the enable status unchanged.

```
void svSYS_InterruptUnBind(
   svSYS_SystemElement *  pPeriph,
   UWORD32                IntIndex);
```

The *pPeriph* parameter specifies the peripheral descriptor of the peripheral generating the interrupt. The *IntIndex* parameter specifies the interrupt number in the peripheral's list of interrupts.

svSYS_InterruptAllBind()
svSYS_InterruptAllUnBind()

These functions will bind all the interrupt sources specified for a peripheral onto a specified interrupt handler routine and enable the associated interrupts, or unbind these interrupts.

```
void svSYS_InterruptAllBind(
   svSYS_InterruptHandler   Handler,
   svSYS_SystemElement    * pPeriph);
void svSYS_InterruptAllUnBind(
   svSYS_SystemElement * pPeriph);
```

The *Handler* parameter specifies the interrupt handler routine to be run when any of the interrupts occur. The *pPeriph* parameter specifies the peripheral descriptor of the peripheral generating the interrupts.

svSYS_InterruptEnable()
svSYS_InterruptDisable()

These functions enable or disable an individual interrupt.

```
void svSYS_InterruptEnable(
   svSYS_SystemElement *  pPeriph,
   UWORD32                IntIndex);
void svSYS_InterruptDisable(
   svSYS_SystemElement *  pPeriph,
   UWORD32                 IntIndex);
```

The *pPeriph* parameter specifies the peripheral descriptor of the peripheral generating the interrupt. The *IntIndex* parameter specifies the interrupt number in the peripheral's list of interrupts.

svSYS_InterruptAllEnable()
svSYS_InterruptAllDisable()

These functions will enable or disable all the interrupt sources specified for a peripheral.

```
void svSYS_InterruptAllEnable(
   svSYS_SystemElement *  pPeriph);
void svSYS_InterruptAllDisable(
   svSYS_SystemElement *  pPeriph);
```

The *pPeriph* parameter specifies the peripheral descriptor of the peripheral generating the interrupt.

Software-XVC Connectivity

For some test scenarios, it may be required to synchronize the execution of software routines with specific hardware events. For example, it may be necessary to observe a software stack in response to data packets being driven into a system peripheral. The software framework, rather than driving the entire system test, can act as a slave waiting for the XVC manager to send commands via a software-interface XVC.

Two global variables are used to hold a pair of base addresses through which data from a software-interface XVC can be read, and to where data from the software verification framework can be written.

```
UWORD32 svSYS_DebugXVCBase;
UWORD32 svSYS_DebugXVCSWBase;
```

Whenever information or commands need to be exchanged between a software-interface XVC and the software, the base addresses are used to monitor requests and response activity between the XVC and the software environment. The exact communication protocol and functions are defined by the XVC and the software designer.

INDEX

ABOUT THE AUTHORS

Janick Bergeron is a Scientist at Synopsys, Inc. He is the author of the best-selling book *Writing Testbenches: Functional Verification of HDL Models* and the moderator of the Verification Guild. He holds a Masters degree in Electrical Engineering from the University of Waterloo, a Bachelor of Engineering from the Université du Québec à Chicoutimi, and a MBA degree from the University of Oregon.

Eduard Cerny, Ph.D. (McGill University), is a Principal Engineer, R&D, in the Verification Group at Synopsys, Inc. He joined Synopsys in 2001 after 25 years in academia, as Professor of Computer Science at the Université de Montréal. His interests have been in design, verification and test of hardware, and he is author of many articles in these areas.

Alan Hunter, BEng(Hons), MSc, is the Design Verification Methodology Programme manager at ARM Ltd. and leads the design verification methodology work for ARM worldwide. This work covers all areas from CPU design verification through systems and system component design verification. His main areas of interest include optimizing design verification efficiency and quality, formal methods, and determinism in the design verification flow.

Andrew Nightingale, BEng(Hons), MBCS CITP, is a consultant engineer at ARM Ltd and has led the SoC Verification group in ARM's Cambridge and Sheffield design centers for several years. The group covers ARM PrimeXSys platforms and PrimeCell development, including advanced AXI- and AHB-based system backplane components such as bus interconnects and high-performance memory controllers.