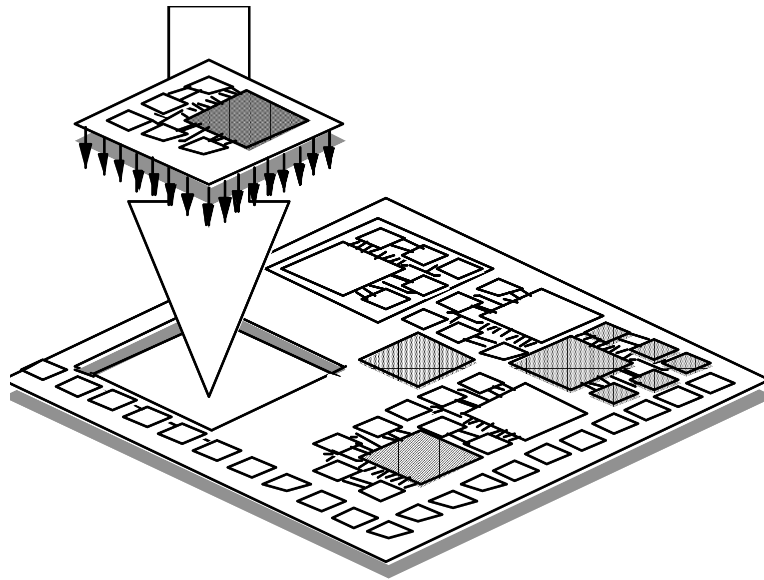
VSI Alliance™

Specification for VC/SoC Functional Verification
Version 1.0

(VER 2 1.0)

Functional Verification
Development Working Group

Released March 2004



Dedication to Public Domain

VSI Alliance hereby dedicates all copyright that VSI Alliance holds in this ______ (the "Work") to the public domain, free of charge, and for the general benefit of the public at large.

VSI Alliance intends this dedication to be an overt act of relinquishment in perpetuity of all present and future rights that VSI Alliance may have in the Work under copyright law, whether vested or contingent, including without limitation, the right to prevent others from freely reproducing, distributing, transmitting, using, modifying, building upon or otherwise exploiting the Work for any purpose, commercial or non-commercial, or in any way.

VSI Alliance understands that such relinquishment includes the relinquishment of all rights to enforce (by lawsuit or otherwise) any copyrights that VSI Alliance may have in the Work.

IMPORTANT - NO WARRANTY. THE WORK IS PROVIDED "AS IS", "WHERE-IS", WITHOUT WARRANTY OF ANY KIND. WITHOUT LIMITING THE GENERALITY OF THE FOREGOING, VSI ALLIANCE EXPRESSLY DISCLAIMS ALL WARRANTIES WITH RESPECT TO THE WORK, WHETHER EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, WARRANTIES OF TITLE, NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, AND IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Functional Verification Development Working Group
(VER 2 1.0)

Company Members of the Development Working Group:

Cadence Design Systems
Elixent
Hewlett-Packard
IBM
Infineon
Intel
Mentor Graphics
Motorola
Palmchip
Synopsys
Verisity Design

Active Contributors:

Tom Anderson (Chair) .... Individual Member
Jeff Barkley .... Individual Member
Mike Bartley .... Elixent
Robin Bhagat .... Palmchip
Robert Birch .... Individual Member
Sherri Bishop .... Mentor Graphics
Thomas Borgstrom .... Individual Member
Annette Bunker .... Individual Member
Steve Burchfiel .... Individual Member
John Emmitt .... Individual Member
Dave Goldberg .... Hewlett-Packard
Kamal Hashmi .... Individual Member
Yaron Kashai .... Verisity Design
Srinivas Madaboosi .... Individual Member
Adriana Maggiore .... Individual Member
Mehdi Mohtashemi .... Synopsys
Mukund Patel .... Intel
Peter Paterson .... Cadence Design Systems
Mark Peryer .... Mentor Graphics
Andrew Piziali .... Verisity Design
Richard Raimi .... Individual Member
Prakash Rashinkar .... Cadence Design Systems
G. David Roberts .... IBM
Mandayam Srivas .... Individual Member
Richard Stolzman .... Individual Member

Revision History

Version 1.0	Jan04	Editorial Staff	Incorporated edits and formatted document

Table of Contents

1. **Overview** 1
1.1 Scope 1
1.2 Organization of Document 1
1.3 Referenced IP 1
1.4 Summary of Deliverables 3
1.4.1 Key to Deliverables 3
1.4.2 Definition of Columns in Summary of Deliverables Table 3
1.4.3 Summary of Deliverables Table 3
2. **Functional Verification Deliverables** 7
2.1 Overview 7
2.2 Documentation 7
2.3 Testbenches 7
2.3.1 Overview 7
2.3.2 Formats 8
2.3.3 Applicability 8
2.4 Drivers 8
2.4.1 Overview 8
2.4.2 Formats 8
2.4.3 Applicability 9
2.5 Monitors 9
2.5.1 Overview 9
2.5.2 Formats 9
2.5.3 Applicability 9
2.6 Assertions 9
2.6.1 Overview 9
2.6.2 Formats 10
2.6.3 Applicability 10
2.7 Functional Coverage 10
2.7.1 Overview 10
2.7.2 Formats 10
2.7.3 Applicability 10
2.7.4 Input Data Functional Coverage 10
2.7.5 Output Data Functional Coverage 11
2.7.6 Internal Data Functional Coverage 11
2.7.7 Input Temporal Functional Coverage 11
2.7.8 Output Temporal Functional Coverage 11
2.7.9 Internal Temporal Functional Coverage 11
2.7.10 Input/Output Temporal Functional Coverage 11
2.8 Code Coverage 11
2.8.1 Overview 11
2.8.2 Formats 11
2.8.3 Applicability 12

2.17 Functional Verification Certificate 21
2.17.1 Formats 22
2.17.2 Applicability 22

3. Reuse of Functional Verification Deliverables in SoC Verification .23

3.1 Overview 23
3.2 VC Re-Verification 23
3.3 VC Re-Verification in an SoC 24

4. Functional Verification Deliverables Rules 25

4.1 Drivers 25
4.1.1 Rules 25
4.1.2 Coding Guidelines 25
4.2 Monitors 26
4.2.1 Rules 26
4.2.2 Coding Guidelines 27
4.3 Assertions 29
4.3.1 Rules 29
4.4 Functional Coverage 29
4.4.1 Input Data Functional Coverage 29
4.4.2 Output Data Functional Coverage 29
4.4.3 Internal Data Functional Coverage 30
4.4.4 Input Temporal Functional Coverage 30
4.4.5 Output Temporal Functional Coverage 31
4.4.6 Internal Temporal Functional Coverage 32
4.4.7 Input/Output Temporal Functional Coverage 32
4.5 Code Coverage 32
4.5.1 User-Defined Metrics for Code Coverage Measurement 32
4.5.2 Toggle Coverage 33
4.5.3 Statement Coverage 33
4.5.4 Branch Coverage 33
4.5.5 Condition Coverage 34
4.5.6 Finite State Machine (FSM) Coverage 34
4.6 Formal Methods 35
4.6.1 Verification Using Equivalence Checking 35
4.6.2 Equivalence Checking Data 35
4.6.3 Transistor or Gate-Level Equivalence Checking 36
4.6.4 Dynamic Formal Methods 36
4.6.5 Formal Coverage 36
4.6.6 Theorem Proving 36
4.6.7 Model Checking and Property Checking 37
4.6.8 Formal Constraint-Driven Stimulus Generation 37
4.6.9 Symbolic Simulation 38

4.7 Documentation 38
4.7.1 Provider Functional Verification Documentation. 38
4.7.2 Functional Verification Deliverable Documentation 41
4.8 Behavioral Models 43
4.9 Scripts 43
4.9.1 Simulation (Verification) Scripts 43
4.10 Stub Model 44
4.11 Functional Verification Certificate 45

5. Testbench Coding Guidelines. 47

5.1 Coding for Verification 47
5.2 General 47
5.3 Symbolic Constants. 48
5.4 Routines. 48
5.5 Signal State and Time 49
5.6 Clocking 50
5.7 I/O and Pads 54
5.8 Messages 54
5.9 Termination 55
5.10 Synchronization. 56
5.11 External Interface Functions (Such as PLI Routines) 57
5.12 Configuration Control 57
5.13 VC Reset 58
5.14 Testbench Interface 58
5.15 Behavioral Models for Memory 59
5.16 Memory Operation 59
5.17 Detailed Behavioral Models for I/O Pads 60
5.18 Stimulus. 60
5.18.1 Random. 60
5.19 Checking 61
5.20 Partitioning 61
5.21 Naming 62
5.22 Memory Map Control 62
5.23 Stimulus Source Code 63

A. Glossary 65

List of Figures

Figure 1: Testbench Architecture and Stimulus Flow 8
Figure 2: Example of VSI FV Rule 4.4.2 30
Figure 3: Example of VSI FV Rule 4.4.4 31
Figure 4: Example of VSI FV Rule 5.23.4 64

List of Tables

Table 1: Summary of Deliverables 3
Table 2: Example of VSI FV Rule 4.4.1 29
Table 3: Example of VSI FV Rule 4.4.5 31
Table 4: Example of VSI FV Rule 4.4.7 32

1. Overview

1.1 Scope

The primary goal of virtual component (VC) functional verification is to establish confidence that the design intent for the VC has been captured correctly and preserved in the implementation. For the end user (integrator) of a VC, gaining confidence that reasonable steps have been taken to verify the VC ultimately affects the decision to use it in a project. For the provider of a VC, a good functional verification methodology should result in tangible commercial benefits such as increased customer satisfaction and lower support and maintenance costs.

There are many different approaches to functional verification, and there are many methods that can be used. Typically, some combination of verification approaches is used in order to achieve the required level of quality. The purpose of this specification is to define a number of common verification approaches, and to indicate how their use by a VC provider could be understood, or even reproduced, by a VC integrator.

Ultimately, the goal of this specification is to facilitate the reuse of as much as possible of the verification environment delivered by the VC provider by the end user in the integration and chip level verification of the end product. The verification environment comprises testbenches, models, scripts and other deliverables that will be discussed in detail.

Given the variety of valid approaches to functional verification, this specification should not be interpreted as mandating the use of all the methods described. The intention is that when a method is used, the appropriate sections of the specification should be applied.

1.2 Organization of Document

The current chapter provides general information on this specification. Chapter 2 discusses the provider functional verification performed by the VC provider and identifies specific deliverables related to functional verification from the VC provider to the VC integrator. It also covers deliverables, including documentation requirements, which facilitate VC integration and SoC verification. Chapter 3 discusses some issues related to verification reuse by the VC integrator, Chapter 4 presents an extensive set of rules for the functional verification deliverables and the final chapter presents a set of guidelines for verification testbench coding, many of which are intended to foster reuse.

1.3 Referenced IP

This specification references a number of data formats and other standards, including:

- Verilog: IEEE 1364-2001
  - Owner: IEEE
  - Status: Accredited standard
  - http://standards.ieee.org/faqs/order.html

- Verilog: IEEE 1364-1995
  - Owner: IEEE
  - Status: Accredited standard, older version
  - http://standards.ieee.org/faqs/order.html

- VHDL: IEEE 1076-1987
  - Owner: IEEE
  - Status: Accredited standard
  - http://standards.ieee.org/faqs/order.html

- OVL: OVL Reference Manual, June 2003
  - Owner: Accellera
  - Status: Standard library
  - http://www.verificationlib.org

- PSL: Property Specification Language Reference Manual Version 1.01
  - Owner: Accellera
  - Status: Standard property language
  - http://www.accellera.org/pslv101.pdf

- SystemVerilog 3.1: Accellera's Extensions to Verilog
  - Owner: Accellera
  - Status: Standard language
  - http://www.eda.org/sv-ec/SystemVerilog_3.1_final.pdf

- SRS: Functional Verification Semiconductor Reuse Standard V3.0
  - Owner: Motorola
  - Status: Industry standard
  - http://www.motorola.com

- VSI: Taxonomy of Functional Verification for Virtual Component Development and Integration (VER 1 1.1)
  - Owner: VSI Alliance
  - Status: Taxonomy
  - http://www.vsi.org/resources/specs/ver111.pdf

- AMBA: IHI-0011A - AMBA Specification Rev2.0
  - Owner: ARM, Ltd.
  - Status: Industry standard
  - http://www.arm.com/armtech/AMBA_Spec

- *e*: IEEE 1647 Draft Standard for the Functional Verification Language '*e*'
  - Owner: IEEE
  - Status: IEEE draft standard

1.4 Summary of Deliverables

These deliverables include the definition of the data representation requirements, selection of the data representation formats, documentation requirements and the definition of appropriate VC guidelines as required for the functional verification of VCs and system-on-chip (SoC) designs.

1.4.1 Key to Deliverables

M - Mandatory

CM - Conditionally Mandatory

R - Recommended

CR - Conditionally Recommended

1.4.2 Definition of Columns in Summary of Deliverables Table

Section - The section number referenced

Deliverable - A summary of the deliverable

Currently Used Formats - Standards used in current design and verification flows

VSIA Specified Format - The VSIA endorsed and specified format; where two formats are specified, both are required

SI Hard - The level of requirement

Comply - Enables the user to check off requirements when using this table

Comments - The conditions for the relevant deliverables

1.4.3 Summary of Deliverables Table

Table 1: Summary of Deliverables

Section	Deliverable	Currently Used Formats	VSIA Specified Format	Rules Sections	SI Hard	Comply	Comments
2.3	Testbenches	Verilog, VHDL, C, C++, Vera, *e*	None	5.1-5.23	M		
2.4	Drivers	Verilog, VHDL, C, C++, Vera, *e*	None	4.1	R		
2.5	Monitors	Verilog, VHDL, C, C++, Vera, *e*, PSL and SystemVerilog	None	4.2	R		
2.6	Assertions	Verilog, VHDL, C, C++, Vera, *e*, PSL and SystemVerilog	None	4.3	R		
2.7.4	Input Data Functional Coverage	Text, Table	None	4.4.1	M		
2.7.5	Output Data Functional Coverage	Text, Table	None	4.4.2	M		
2.7.6	Internal Data Functional Coverage	Text, Table	None	4.4.3	M		

Table 1: Summary of Deliverables (Continued)

Section	Deliverable	Currently Used Formats	VSIA Specified Format	Rules Sections	SI Hard	Comply	Comments
2.7.7	Functional Coverage	Text, Table	None	4.4.4	CM		
2.7.8	Output Temporal Functional Coverage	Text, Table	None	4.4.5	CM		
2.7.9	Internal Temporal Functional Coverage	Text, Table	None	4.4.6	CM		
2.7.10	Input/Output Temporal Functional Coverage	Text, Table	None	4.4.7	CM		
2.8	Code Coverage	Microsoft Word, PDF, HTML	None	4.5	R		
2.9.2	Equivalence Checking	None	None	4.6.1	M		
2.9.3	Equivalence Checking Data	None	None	4.6.2	CR		
2.9.4	Transistor or Gate-Level Equivalence Checking	None	None	4.6.3	CR		
2.9.5	Dynamic Formal Methods	None	None	4.6.4	CR		
2.9.6	Formal Coverage	None	None	4.6.5	CR		
2.9.7	Theorem Proving	None	None	4.6.6	CR		
2.9.8	Model Checking and Property Checking	None	None	4.6.7	CR		
2.9.9	Formal Constraint-Driven Stimulus Generation	None	None	4.6.8	CR		
2.9.10	Symbolic Simulation	None	None	4.6.9	CR		
2.10.1	Provider Functional Verification Documentation	Microsoft Word, PDF	None	4.7.1	M		
2.10.2	Functional Verification Deliverable Documentation	Microsoft Word, PDF	None	4.7.2	M		
2.11	Behavioral Models	Verilog, VHDL, C, C++, Vera, *e*	None	4.8	CM		
2.12	Behavioral Models for Memory	Verilog, VHDL, C, C++, Vera, *e*	None	5.15	CR		
2.13	Detailed Behavioral Models for I/O Pads	Verilog, VHDL, C, C++, Vera, *e*	None	5.17	CM		

Table 1: Summary of Deliverables (Continued)

Section	Deliverable	Currently Used Formats	VSIA Specified Format	Rules Sections	SI Hard	Comply	Comments
		C++, Vera, *e*					
2.15	Scripts	Tcl, Perl, make, UNIX shell	None	4.9	M		
2.16	Stub Model	Verilog, VHDL, C, C++, Vera, *e*	None	4.10	R		
2.17	Functional Verification Certificate	Text	None	4.11	R		

2. Functional Verification Deliverables

2.1 Overview

A VC may be functionally verified by its provider in a number of different ways prior to release to end users. This process is referred to as provider functional verification. This specification covers a number of valid approaches to verifying VC functionality. Any combination might be used by a provider; therefore, only those sections that relate to the techniques used should be followed.

The specification lists rules and guidelines, and then gives justifications and numerous examples. A rule must be followed if the verification method is used. A guideline should be followed if circumstances permit.

2.2 Documentation

The purpose of the provider functional verification documentation is to describe the methodology used to verify the VC by the VC provider and to give some specific information about the test cases used and stimulus applied. This documentation may also include information on the functional coverage achieved, or information on code coverage, thus giving an indication of how well the VC has been verified. This information is supplied to enable VC integrators to make an informed judgment on the quality of the VC or the type of chip-level verification that they should use to achieve their quality goals when they integrate the VC with the rest of their design. The type of documentation required is a by-product of any reasonable functional verification methodology.

The functional verification methodology used to validate the VC encompasses the overall strategy used by the VC provider. Examples include the functional verification environment, the functional tests run, functional or code coverage achieved, any assertion methods used, any standards compliance test suites run, any formal property checks made, any validation run in a hardware platform, or any other functional verification technique used.

2.3 Testbenches

2.3.1 Overview

The term testbench can be used to describe a wide range of implementation complexity, ranging from a very basic set of pin-level stimulus run standalone in a simulator to the major part of a sophisticated functional verification environment containing complex models and interfaces to pattern generation tools which are run in conjunction with the simulator.

The testbench deliverable consists of the components required to exercise the VC using traditional simulation methods that identify differences between expected and actual behavior. The testbench components are part of a structured verification approach that exercises the VC with transaction-based stimulus. Typically, the testbench contains a common set of routines used to clock, reset, and synchronize stimulus as it pertains to driving the VC in a simulation environment. The testbench also includes the instantiation of the VC itself, along with an appropriate mix of functional models, drivers, monitors, and stimulus code.

Coding guidelines for testbenches are outlined in Chapter 5.

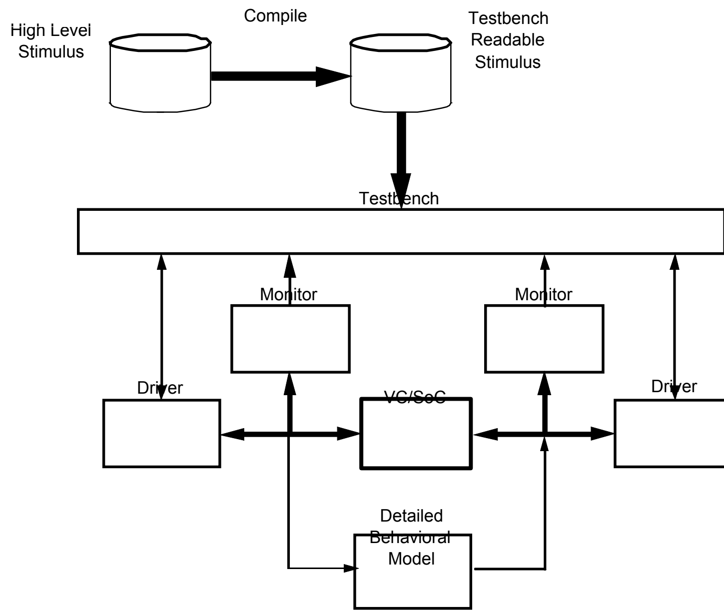


Figure 1: Testbench Architecture and Stimulus Flow

2.3.2 Formats

There is no restriction on the format for the delivery. Example formats for the delivery include Verilog, VHDL, C, C++, Vera, and *e*.

2.3.3 Applicability

- Soft VCs: Mandatory
- Firm VCs: Mandatory
- Hard VCs: Mandatory

This deliverable is mandatory for soft, firm and hard VCs for the following reasons:

- The VC integrator needs to be able to verify the installation of the VC data.
- The ability to run simulations and observe behavior aids in the knowledge transfer of the VC.

2.4 Drivers

2.4.1 Overview

Drivers are the part of the testbench responsible for stimulating the signals of the VC and capturing the signal responses from the VC. They have two interfaces:

- The VC side which connects the driver to the VC
- The testbench side which connects the driver to the testbench

These two interfaces might operate at different levels of abstraction. The VC-side interface must always operate at the same abstraction level as the simulation model of the VC (for example, the signal level for RTL). The testbench side might operate at a higher level of abstraction such as the transaction level. The driver must act as a bridge between those two levels.

2.4.2 Formats

There is no restriction on the format for the delivery. Example formats for the delivery include Verilog, VHDL, C, C++, Vera, and *e*.

2.4.3 Applicability

- Soft VCs: Recommended
- Firm VCs: Recommended
- Hard VCs: Recommended

It is recommended that drivers be a separate deliverable within the testbench for soft, firm, and hard VCs. This is not required to perform functional verification, but it does add significantly to the quality and portability of the verification. For example, VC integrators could reuse it in their own testbench.

2.5 Monitors

2.5.1 Overview

Monitors are probes that watch signals in the VC. The probes can be used for various purposes, for example:

- Protocol verification: ensuring that the interface signals obey the protocol defined for the interface
- Performance verification: ensuring that the interface signals obey the performance targets defined for the VC (in terms of cycle counts rather than intra-cycle measures or time intervals)
- Recording functional coverage data

Monitors may be split into two types:

- Interface monitors that only monitor VC interface signals
- Internal monitors that only monitor signals internal to the VC

The monitors should be usable at both the VC verification level and at the SoC level (to ensure that the VC is working correctly within the VC integrator's system).

Monitors may be explicitly coded, for example protocol monitors on interfaces, but they can also arise from assertions—statements of intent—in the design. Although assertions are most strongly associated with formal verification, they are also very useful in simulation. Virtually all assertion languages and libraries come with a mechanism to convert the assertions into monitors (also called checkers) for simulation.

2.5.2 Formats

There is no restriction on the format for the delivery. Example formats for the delivery include Verilog, VHDL, C, C++, Vera, *e*, PSL and SystemVerilog.

2.5.3 Applicability

- Soft VCs: Recommended
- Firm VCs: Recommended
- Hard VCs: Recommended

Interface monitors are a recommended deliverable for soft, firm, and hard VCs. They are not required to perform functional verification, but they add significantly to the quality of the verification

2.6 Assertions

2.6.1 Overview

Assertions are statements of design intent that usually can be leveraged in both simulation and formal verification. They are specified by the design engineer as part of the design capture or by the verification engineer to make the design more verifiable and reusable.

Assertions can be specified in many ways. For example, if the VC is modeled in VHDL, checkers can be written using the "assert" statement. Similarly, checkers for Verilog and VHDL can be written using OVL. Especially for a soft VC, providing the assertions in their original specification form can aid in understanding the design. The assertions may be specified in-line within the VC RTL, within the testbench code or in a separate file.

Even if the assertions are not provided, it is recommended that any simulation monitors derived from them be provided by the VC integrator. Deliverables requirements related to the use of assertions in simulation are covered in the section on monitors. If the assertions are also used in formal verification (such as model checking, property checking or dynamic formal verification), the deliverables requirements in the respective sections apply.

2.6.2 Formats

There is no restriction on the format for the delivery. Example formats for the delivery include Verilog, VHDL, C, C++, Vera, *e*, OVL, PSL and SystemVerilog.

2.6.3 Applicability

- Soft VCs: Recommended
- Firm VCs: Recommended
- Hard VCs: Recommended

Assertions in specification form are a recommended deliverable for soft, firm, and hard VCs. They are not required to perform functional verification, but they add significantly to the quality of the verification

2.7 Functional Coverage

2.7.1 Overview

A quantitative measure of how well a VC has been verified may be specified through functional coverage measurement. The functional requirements of a VC are represented in a set of coverage models. The coverage models described in the following sections are organized in two dimensions whose values are: input/output/internal and data/temporal.

Input coverage models record characteristics of the stimulus stream applied to the VC. Output coverage models record characteristics of the response of the VC to applied stimulus. Internal coverage models record behavior internal to the VC.

Data coverage models record characteristics of control and data values. Temporal coverage models record time-based behavior of measured interfaces.

2.7.2 Formats

There is no restriction on the format for the delivery; functional coverage information will generally be provided in text or tabular form in one of the following formats:

(a) Multi-dimensional organization of attributes

(b) Hierarchical structure of attributes

(c) Hybrid of (a) and (b)

2.7.3 Applicability

- Soft VCs: Conditionally Mandatory
- Firm VCs: Conditionally Mandatory
- Hard VCs: Conditionally Mandatory

As described in the following sections, functional coverage metrics are a mandatory deliverable for soft, firm, and hard VCs when protocols are sufficiently complex. They help the integrator assess the quality of the VC verification.

2.7.4 Input Data Functional Coverage

Input data functional coverage records value parameters of the stimulus applied to the VC. The VC provider must provide measurements of input data functional coverage with the VC.

2.7.5 Output Data Functional Coverage

Output data functional coverage records value parameters of the response of the VC to applied stimulus. The VC provider must provide measurements of output data functional coverage with the VC.

2.7.6 Internal Data Functional Coverage

Internal data functional coverage measurements record observed behavior of the design within the data domain. For example, the values written into a control register, FIFO depth, and a linked list pointer are internal values. The VC provider must provide measurements of internal data functional coverage with the VC.

2.7.7 Input Temporal Functional Coverage

Input temporal functional coverage records temporal behavior of the stimulus applied to the VC. If at least one input interface to the VC uses a temporal protocol more complex than, for example, periodic checking, the VC provider must provide measurements of temporal functional coverage for each such interface.

2.7.8 Output Temporal Functional Coverage

Output temporal functional coverage records the temporal response of the VC to the applied stimulus. This temporal response is a characteristic of, for example, a protocol. If at least one output interface to the VC uses a temporal protocol more complex than, for example, periodic checking, the VC provider must provide measurements of temporal functional coverage for each such interface.

2.7.9 Internal Temporal Functional Coverage

Internal temporal functional coverage measurements record observed behavior of the design within the time domain. For example, the handshake between an instruction prefetcher and decoder would define a protocol that could be described as temporal coverage. The VC provider must provide measurements of internal temporal functional coverage if there are temporal protocols of sufficient complexity.

2.7.10 Input/Output Temporal Functional Coverage

Input/output temporal functional coverage measurements record the observed behavior of the design between the input and output interfaces within the time domain. For example, a specific packet type that is accepted on an input interface, and is followed five cycles later by a second packet type transmitted from an output interface can be described as temporal coverage. The VC provider must provide measurements of input/output temporal functional coverage if there are temporal protocols of sufficient complexity.

2.8 Code Coverage

2.8.1 Overview

Code coverage measurement metrics are used to assess the extent to which the VC code has been exercised during simulation and to judge the quality of the verification. These metrics are also used as an aid to judge when the simulation verification is nearing completion. Code coverage identifies code structures that have not been verified. By itself, it does not show that the design is functionally correct. The utility of coverage metrics lies in pointing out functional code that has not been exercised in simulation.

There are several defined metrics used in the industry today. These are (a) toggle coverage, (b) statement coverage, (c) branch coverage, (d) condition coverage, and (e) finite state machine coverage. A brief definition of each metric is given in the following sections. For more information, refer to the VSIA Taxonomy of Functional Verification for Virtual Component Development and Integration.

2.8.2 Formats

There are no specified formals for code coverage metrics; they may be provided in such format as Microsoft Word, PDF, HTML or text.

2.8.3 Applicability

- Soft VCs: Recommended
- Firm VCs: Recommended
- Hard VCs: Recommended

Code coverage metrics are a recommended deliverable for soft, firm, and hard VCs. They are not required to perform functional verification, but they help to assess the quality of the verification

2.8.4 User-Defined Metrics for Code Coverage Measurement

There are a number of user-defined metrics used in the industry and various EDA tools are available to measure code coverage. The rules associated with such metrics are defined in the next sections.

2.8.5 Toggle Coverage

Toggle coverage is a form of code coverage that checks whether each node has changed its polarity (0 to 1 and 1 to 0) during simulation.

2.8.6 Statement Coverage

Statement coverage is a form of code coverage that checks which executable statements in the RTL source code were executed during simulation.

2.8.7 Branch Coverage

Branch coverage is a form of code coverage that checks which branches in the RTL source code were taken during simulation.

2.8.8 Condition Coverage

Condition coverage is a form of code coverage that examines the sub-expressions in condition statements to determine which values of those sub-expressions caused higher-level expressions to be true or false during simulation.

2.8.9 Finite State Machine (FSM) Coverage

FSM coverage is a form of code coverage that reports which states, transitions, and paths of an FSM were simulated.

2.8.10 Code Coverage Report

The RTL (VHDL or Verilog) code coverage report documents the summary of the overall code coverage achieved for branch, statement, FSM, and condition-type metrics. It also reports on code structures that cannot be exercised, thereby reducing coverage, and briefly explains why they cannot be exercised.

2.9 Formal Methods

2.9.1 Equivalence Checking

Formal logic equivalence checking applies formal verification techniques to compare two designs for functional equivalence. The designs must be similar but may have been implemented differently, for example RTL and gate-level logic. Formal logic equivalence checking capability has progressed to the extent that logic equivalence checking tools (both commercial and internally developed) are available and are capable of comparing industrial designs more quickly and accurately than by using simulation. It is therefore important that a VC be verifiable using commercial equivalence checkers in the integrator's environment. VC providers must use equivalence checking in their design process when they deliver firm and hard VCs.

2.9.2 Verification Using Equivalence Checking

All VCs that can be modified by the integrator by any method that changes the netlist or layout must allow equivalence checking between the unaltered soft VC and the modified version. Even if the VC provider does not use equivalence checking, it must be possible for the VC integrator to do so for soft and firm VCs.

2.9.2.1 Formats

The format of the data is specific to the tool or method used.

2.9.2.2 Applicability

- Soft VCs: Mandatory
- Firm VCs: Mandatory
- Hard VCs: N/A

2.9.3 Equivalence Checking Data

It is recommended that comprehensive formal logic equivalence checking be performed between the fully verified design and the VC, if a different view is provided to the integrator. It is optional for a VC provider to use equivalence-checking methods as a part of its verification.

2.9.3.1 Formats

The format of the data is specific to the tool or method used.

2.9.3.2 Applicability

- Soft VCs: Conditionally Recommended
  If the provider employs equivalence checking as part of the standalone verification process for a soft VC, it is recommended that documentation be provided to the integrator.
- Firm VCs: Conditionally Recommended
  If the provider employs equivalence checking as part of the standalone verification process for a firm VC, it is recommended that documentation be provided to the integrator.
- Hard VCs: Conditionally Recommended
  If the provider employs equivalence checking as part of the standalone verification process for a hard VC, it is recommended that documentation be provided to the integrator.

2.9.4 Transistor or Gate-Level Equivalence Checking

Tools are available that allow logical information to be extracted from transistor or switch-level netlists for use with formal equivalence checking tools. The current commercial tools generate an HDL netlist that can be used by the equivalence checker.

2.9.4.1 Formats

The format of the data is specific to the tool or method used.

2.9.4.2 Applicability

- Soft VCs: Conditionally Recommended
  If the provider employs equivalence checking as part of the standalone verification process for a soft VC, it is recommended that documentation be provided to the integrator.
- Firm VCs: Conditionally Recommended
  If the provider employs equivalence checking as part of the standalone verification process for a firm VC, it is recommended that documentation be provided to the integrator.
- Hard VCs: N/A
  If no editable view is provided, equivalence checking cannot be performed.

2.9.5 Dynamic Formal Methods

Dynamic formal verification uses formal, mathematical methods to amplify or expand design behavior exercised in simulation. Like static formal verification, it targets assertions by considering a wide range of behaviors that conform to any input constraints. It does not necessarily start from the reset state and consider all possible behaviors for all time. Instead, it starts from a series of states already reached in simulation and explores a range of behavior around that state, usually bounded by sequential depth (the number of clocks). For example, dynamic formal verification may consider all possible five-cycle sequences of legal input changes and associated state transitions from each state in a simulation trace. This technique is optimized for finding ways to violate assertions; it is unlikely to prove that assertions can never be violated.

It is optional for a VC provider to use dynamic formal methods as part of the standalone VC verification. Although optional for all categories, deliverables related to dynamic formal verification are more likely to be provided for a soft VC, since the assertions and constraints often reference source signals.

2.9.5.1 Formats

There are no applicable standard formats for dynamic formal verification documentation. Any information provided is tied to a specific vendor tool.

2.9.5.2 Applicability

- Soft VCs: Conditionally Recommended
  If the provider employs dynamic formal verification as part of the standalone verification process for a soft VC, it is recommended that documentation be provided to the integrator.
- Firm VCs: Conditionally Recommended
  If the provider employs dynamic formal verification as part of the standalone verification process for a firm VC, it is recommended that documentation be provided to the integrator.
- Hard VCs: Conditionally Recommended
  If the provider employs dynamic formal verification as part of the standalone verification process for a hard VC, it is recommended that documentation be provided to the integrator.

2.9.6 Formal Coverage

Formal coverage, sometimes called semi-formal verification, refers to the use of static or dynamic formal methods to improve coverage results as measured by some appropriate metric. Usually, this is accomplished by placing assertions on points not covered in the existing simulation tests, and then targeting these assertions with formal methods. For example, an assertion could be placed inside a basic block in RTL to improve line coverage, or on a state machine to improve arc coverage.

Formal coverage is an emerging verification methodology that is neither widely employed nor directly supported by many formal verification tools.

It is optional for the VC provider to employ formal methods to improve coverage.

2.9.6.1 Formats

There are no applicable standard formats for formal coverage documentation. Any information provided is tied to a specific vendor tool.

2.9.6.2 Applicability

- Soft VCs: Conditionally Recommended
  If the provider employs formal coverage improvement as part of the standalone verification process for a soft VC, it is recommended that documentation be provided to the integrator.
- Firm VCs: Conditionally Recommended
  If the provider employs formal coverage improvement as part of the standalone verification process for a firm VC, it is recommended that documentation be provided to the integrator.
- Hard VCs: Conditionally Recommended
  If the provider employs formal coverage improvement as part of the standalone verification process for a hard VC, it is recommended that documentation be provided to the integrator.

2.9.7 Theorem Proving

Theorem proving is a formal verification technique in which the system specification and the system implementation are represented in a formal logical system. Relationships between the two are verified using inference rules, rewrites, and decision procedures. Logics used for theorem-proving activities are typically undecidable. Theorem proving generally requires a large amount of user guidance.

It is optional for a VC provider to use theorem-proving methods as a part of its verification effort. Although optional for all categories, deliverables related to theorem proving are more likely to be provided for soft VCs, since the relationships proven often reference source signals.

2.9.7.1 Formats

The documentation format depends on the logic and tools that are used.

2.9.7.2 Applicability

- Soft VCs: Conditionally Recommended
  If the provider employs theorem proving as part of the standalone verification process for a soft VC, it is recommended that documentation be provided to the integrator.
- Firm VCs: Conditionally Recommended
  If the provider employs theorem proving as part of the standalone verification process for a firm VC, it is recommended that documentation be provided to the integrator.
- Hard VCs: Conditionally Recommended
  If the provider employs theorem proving as part of the standalone verification process for a hard VC, it is recommended that documentation be provided to the integrator.

2.9.8 Model Checking and Property Checking

Model checking algorithmically checks whether a model satisfies a specification by exhaustively exploring the reachable state space and testing the truth of the specification at each state. The specification is usually expressed in a temporal logic. The model may be expressed as a directed graph in which a set of atomic propositions is associated with each node. The nodes represent states of the design, and the edges represent possible state transitions, while the atomic propositions represent the basic properties that hold at each state.

For the purposes of this specification, the term property checking is assumed to be equivalent to the term model checking.

Commercial and non-commercial tools exist that perform static functional verification of HDL designs using model- and property-checking techniques. In application, the user selects or specifies functional behaviors to be verified, and then invokes the tool to prove them for a given HDL design.

The scope of application includes static functional verification of one or more of the following:

- Behaviors that can be automatically extracted and assumed using the hardware design language specification and functional design rules.
- Functional checkers that are specified by the designer and that are eligible targets for formal verification.
- Designer-specified functions or behaviors for formal verification. These vary from basic to complex.
- Industry-standard functions such as on-chip bus protocols.

Static functional verification using model checking or property checking is optional. Model checking of critical design behaviors using static functional verification is highly recommended.

2.9.8.1 Formats

Documentation format depends on the logic and tools that are used.

2.9.8.2 Applicability

- Soft VCs: Conditionally Recommended
  If the provider employs model checking or property checking as part of the standalone verification process for a soft VC, it is recommended that documentation be provided to the integrator.

- Firm VCs: Conditionally Recommended
  If the provider employs model checking or property checking as part of the standalone verification process for a firm VC, it is recommended that documentation be provided to the integrator.
- Hard VCs: Conditionally Recommended
  If the provider employs model checking or property checking as part of the standalone verification process for a firm VC, it is recommended that documentation be provided to the integrator.

2.9.9 Formal Constraint-Driven Stimulus Generation

Formal constraint-driven stimulus generation is the utilization of formal methods to generate targeted tests that satisfy a given set of constraints. A set of constraints restricts the behavior of a subset of the VC input signals over time. The constraints are expressed in a formal specification language (for example, temporal logic). Given the constraints, a stimulus generation tool that uses formal techniques (for example, a model checker) calculates a sequence of stimulus for the VC that satisfies the constraints. Depending on the level of integration between the generation tool and the simulator, the stimulus can be directly applied to the VC or saved in a format suitable for simulation. This is an emerging verification methodology that is neither widely employed nor directly supported by many formal verification tools.

It is optional for the VC provider to use formal constraint-driven stimulus generation as part of the standalone VC verification.

2.9.9.1 Formats

There are no applicable standard formats for formal coverage documentation. Any information provided is tied to a specific vendor tool.

2.9.9.2 Applicability

- Soft VCs: Conditionally Recommended
  If the provider employs formal constraint-driven stimulus generation as part of the standalone verification process for a soft VC, it is recommended that documentation be provided to the integrator.
- Firm VCs: Conditionally Recommended
  If the provider employs formal constraint-driven stimulus generation as part of the standalone verification process for a firm VC, it is recommended that documentation be provided to the integrator.
- Hard VCs: Conditionally Recommended
  If the provider employs formal constraint-driven stimulus generation as part of the standalone verification process for a hard VC, it is recommended that documentation be provided to the integrator.

2.9.10 Symbolic Simulation

Symbolic simulation uses the specification of logical symbols to propagate boolean expressions during simulation. By using symbols rather than binary values, functional simulation is performed more efficiently. Symbolic simulation is considered a specialized application of formal technology. The underlying technology uses formal methods to specify, manage, and resolve the behaviors of the symbols. Hence, the formal technology is generally not apparent to the user.

2.9.10.1 Formats

The format of the data is specific to the tool or method used.

2.9.10.2 Applicability

- Soft VCs: Conditionally Recommended
- Firm VCs: Conditionally Recommended
- Hard VCs: Conditionally Recommended
  If the VC provider uses symbolic simulation, it is recommended that the data be provided to the integrator.

2.10 Documentation

2.10.1 Provider Functional Verification Documentation

High-quality functional verification can be achieved in a number of ways. As discussed in previous sections, many techniques are not used by all VC providers and so documentation of these results is recommended. However, the provider functional verification documentation must include the following information for every VC:

- Functional coverage documentation for the provider verification test suite or other process
- Documentation of all third-party test suites used to determine compliance to a standard
- Documentation of all standard test suites used
- Documentation of all hardware test suites used
- Documentation of the tool and platform configuration used
- Documentation of the details of the environment used for VC verification
- Documentation of the design languages used (for example, VHDL or Verilog)
- Documentation of the directory structure used
- Documentation of the testbench files and structure used
- Documentation of the models (memory or other functions) used
- Documentation of the stimulus generator (including drivers) used
- Documentation of the response checker (including monitors) used
- Documentation of the bus functional model (BFM) for on-chip bus (OCB) VCs
- Documentation of all the software drivers used
- Documentation of the design parameterization options and how they were checked

Ideally, this documentation should contain a cross-reference between the VC's functional definition and the tests that exercise that functionality. The list of functional tests that have been carried out on the VC should tie up with the functional definition supplied with the VC, documenting the purpose of each test, the functionality of the test, and the outcome.

The detail supplied in this documentation allows potential end users of the VC to make an informed decision on their level of confidence in its design quality, and to determine whether the functionality they intend to use has been sufficiently exercised. After the decision has been made to use the VC, the information can be used to select specific test cases based on the functionality they check.

2.10.1.1 Formats

- Microsoft Word
- PDF

2.10.1.2 Applicability

- Soft VCs: Mandatory
- Firm VCs: Mandatory
- Hard VCs: Mandatory

2.10.2 Functional Verification Deliverable Documentation

The VC provider must provide a user's guide or an implementer's guide that describes how the VC can be set up, debugged, and used in the end user's environment. This guide must include at least the following sections:

- Documentation of the directory structure used and setup process
- A manifest of all the deliverables and a description of the deliverables, file names (meaningful names), size (in KB), and so on.
- Documentation of how to compile the verification environment.
- Documentation of how to use, set up, and debug the testbench.
- Documentation of the verification steps to be performed by the VC integrator.

- Documentation of how to interpret the results of the deliverable tests.
- Documentation of how certain deliverables could be reused for system-level verification.
- Documentation of any provider tests that cannot be reproduced using the functional verification deliverables.

2.10.2.1 Formats

- Microsoft Word
- PDF

2.10.2.2 Applicability

- Soft VCs: Mandatory
- Firm VCs: Mandatory
- Hard VCs: Mandatory

2.11 Behavioral Models

2.11.1 Overview

Behavioral models of the VC can exist at various levels of abstraction. Such models can be used for the following purposes:

- Verifying VC functionality if the model is a “golden” representation designed to be hooked up in parallel to and compared against the VC
- Enabling verification if the component the model represents is required to be present to adequately test the VC
- Providing a clean simulation (and in the case of analog VCs, a non-mixed mode simulation)
- Speeding up simulation if the model is a representation of the VC
- Increasing the overall testbench portability

The following list gives some examples. It is not intended to be exhaustive.

- Detailed Behavioral Model: A detailed behavioral model is a representation of the function and timing of a component that provides critical implementation details, while abstracting away non-crucial details (such as internal implementation not required by the user).

  For example, in the case of a detailed behavioral model of a CPU, the user can write test cases in a high-level language that can be compiled and then executed by the model. The same test cases can also be compiled and executed on other models of the CPU (such as the HDL model). The justification for such a model is that it is necessary to verify “real” code sequences, actual applications and algorithms where system-level behavior is part hardware and part software. A behavioral model is useful because it usually simulates at much higher speed. It therefore allows the test cases to stress hardware corner cases and pathological timing situations.
- Bus Functional Model. This is a clock or transactional-timing accurate representation of the VC on the external interfaces. It accurately mimics the external behavior and timing of the VC without providing details of the internals. This allows for faster simulation times, while still being useful for SoC integration verification.

For example, the VC might be an on-chip bus master, and the model might provide a transaction-level API. The user can then write test cases at transaction level, specifying the transactions and sequences to be driven. This makes it much easier to create corner cases on interfaces, with finer control of timing and concurrency, and hence efficient deterministic testing of interface logic. The limitation is that this can only be used for integration verification. That is, it doesn't verify system-level functionality.

- Purely functional model. This only models the functional behavior of the VC and can be used as a golden reference for verification purposes. For example, a model of an external memory controller might model the reads and writes to external memories as stores and retrievals from a simple memory. This could be used as a golden reference for the real VC. Similarly, a CPU could be modeled without any pipeline, cache, or other architectural features. However, it could still act as a golden reference for verification (with comparison being performed when the model and VC were expected to be in sync, for example, on instruction retirement).

2.11.2 Formats

There is no restriction on the format for the delivery. Example formats are Verilog, VHDL, C, C++, Vera, and *e*.

2.11.3 Applicability

- Soft VCs: Recommended (unless the delivered verification environment uses it, in which case it is mandatory)
- Firm VCs: Recommended (unless the delivered verification environment uses it, in which case it is mandatory)
- Hard VCs: Recommended (unless the delivered verification environment uses it, in which case it is mandatory)

2.12 Behavioral Models for Memory

2.12.1 Overview

A detailed behavioral model for memory is a representation of the function and timing of a memory while abstracting away non-crucial implementation details.

A high-level behavioral model of memory represents the function of memory but not its timing (although cycle accuracy may still be required). This can be used in a similar way to the detailed behavioral models.

2.12.2 Formats

There is no restriction on the format for the delivery. Example formats are Verilog, VHDL, C, C++, Vera, and *e*.

2.12.3 Applicability

- Soft VCs: Recommended if used in or with VCs
- Firm VCs: Recommended if used in or with VCs
- Hard VCs: Recommended if used in or with VCs

2.13 Detailed Behavioral Models for I/O Pads

2.13.1 Overview

A detailed behavioral model for I/O pads is a representation of the function and timing of the non-standard I/O pads used in a VC.

Such models must be provided under the following situations:

- The I/O pads are non standard and manipulate the signaling on the periphery of the VC
- The I/O pads are the expected connection point of the VC to the user's design
- The I/O pad functionality is required to conduct adequate testing of the VC

2.13.2 Formats

There is no restriction on the format for the delivery. Example formats are Verilog, VHDL, C, C++, Vera, and *e*.

2.13.3 Applicability

- Soft VCs: Mandatory if I/O pads manipulate signals to or from the VC
- Firm VCs: Mandatory if I/O pads manipulate signals to or from the VC
- Hard VCs: Mandatory if I/O pads manipulate signals to or from the VC

2.14 Stimulus

2.14.1 Overview

Stimulus comprises the inputs to the testbench that stimulate the VC within the testbench. It is recommended that it be defined at a high level of abstraction and be translated using the testbench to interface to the VC. This helps readability and portability. This is basically the driver approach.

2.14.2 Formats

There is no restriction on the format for the delivery. Example formats for the delivery are Verilog, VHDL, C, C++, Vera, and *e*.

2.14.3 Applicability

- Soft VCs: Recommended
- Firm VCs: Recommended
- Hard VCs: Recommended

2.15 Scripts

2.15.1 Simulation (Verification) Scripts

The simulation scripts are an integral part of the verification environment deliverables. The scripts provided should encompass the verification environment, regression simulation, and any other scripts that help in setting up and debugging the user's environment. Regression scripts are essential to ensure that patterns can be run in an automated manner.

There are a number of scripts that VC providers should provide for VC integrators to use. The scripts provide help in running simulations and debugging the end user’s setup and environment for the VC. Some of the scripts VC providers should provide are:

- Environment (Types and Partitioning): scripts that support setups for the VC end user.
- Running Regression (in batch mode) test simulations: scripts used for simulating the model and running regression tests, associated documentation, setup (ability to run at VC or SoC level), and issue resolution. It should also support running various testbench configurations, views, and types of patterns.

The scripts provided should be portable and easy to run and adapt by end users in their environment. Scripts are usually implemented as programs written in a UNIX shell language and associated utilities.

2.15.1.1 Formats

There is no restriction on the format for the delivery of simulation scripts. However, some example formats for the scripts provided are: Tcl, Perl, make and UNIX shell.

2.15.1.2 Applicability

- Soft VCs: Mandatory
- Firm VCs: Mandatory
- Hard VCs: Mandatory

2.15.1.3 Environment (Types and Partitioning) Scripts

Environment setup scripts should document the procedure that can be used by the end user of the VC. This is helpful in understanding the setups required, and can further help minimize the debug effort required. The VC supplier must support regressions to be run at the VC level only. The regression environment must also provide the flexibility to run regressions using various testbench configurations, various VC views, running all types of patterns, and a comparison of current simulation results against reference results.

2.15.1.4 Regression (Job Control) Simulation Scripts

Simulation scripts to run regression testing or simulation should be provided. Efficiently running regressions is an important verification task. Regression scripts may be written to automate the task of running regressions and gathering results. Regression scripts should be written in a style that allows them to be run in parallel on multiple machines on the network rather than only serially on a single machine. The regression tests can complete more quickly and the ability to quickly analyze the results enhances the productivity of verification personnel.

2.16 Stub Model

2.16.1 Overview

The stub model is a very simple model that includes only the inputs, outputs or bi-directional signals the VC may have. Specific output values can be assigned. The model is used as placeholder that allows the user to verify the connectivity of testbenches that instantiate the VC.

2.16.2 Formats

There is no restriction on the format for the delivery. Example formats for the delivery are Verilog, VHDL, C, C++, Vera, and *e*.

2.16.3 Applicability

- Soft VCs: Recommended
- Firm VCs: Recommended
- Hard VCs: Recommended

This is a recommended deliverable because it assists the VC integrator. Not all integrators use the stub model. If a VC integrator needs a stub model and one is not provided, it may be created from the top-level VC structure anyway.

2.17 Functional Verification Certificate

The functional verification certificate is a standardized ASCII text file that is generated by the verification environment to document the outcome of the simulations run. This file is generated by the provider as a by-product of the verification process, and is part of the VC deliverables. The end user can then regenerate the certificate after running the deliverable verification environment to confirm the same functionality. The certificate contains the following fields:

- Name and version of VC
- Name and version of VC functional verification environment
- Date simulations run
- Configuration of VC used (for parameterized VCs)
- Name of any tests run
- Outcome of any tests run

The functional verification certificate provides the user with the confidence that the VC delivered is the VC documented, and that it has been tested using the delivered verification environment.

2.17.1 Formats

ASCII text file

2.17.2 Applicability

- Soft VCs: Recommended
- Firm VCs: Recommended
- Hard VCs: Recommended

3. Reuse of Functional Verification Deliverables in SoC Verification

3.1 Overview

This specification encompasses four key concepts in VC and SoC verification:

- The VC provider should follow best practices when performing provider functional verification, including both standalone VC verification and any system-level testing performed.
- The VC provider must thoroughly document the provider functional verification process so that potential customers can make informed decisions before committing to use the VC.
- The VC provider should provide enough components of the standalone VC verification environment for the VC integrator to perform "out-of-box" testing and re-verify the VC after any changes
- The VC provider should provide as much assistance as possible for the VC integrator to re-verify the VC in the context of the complete SoC.

All of the deliverables discussed in the previous two chapters, as well as the rules and guidelines in the following two chapters, are intended to support these concepts and encourage best practices during the functional verification of both the VC and the SoC. This chapter covers a few additional points on reuse of the functional verification deliverables by the VC integrator.

3.2 VC Re-Verification

As summarized in the previous two chapters, the VC provider should deliver many parts of the standalone VC verification environment along with the VC. Although not universal, most major VC providers do follow this recommendation. There are several reasons why the VC integrator may request or even demand that their providers comply.

If the VC integrator can reproduce all, or a significant amount, of the standalone verification then this supports "out-of-box" verification of the newly-acquired VC. This can serve as an incoming quality inspection, allowing the integrator to observe the simulations running and see how the VC behaves. The integrator might also want to repeat certain coverage measurements to double-check the results reported by the provider. If the VC implements a standard interface, it is quite common for integrators to acquire a third-party protocol monitor and include it in the simulations to cross-check the VC provider's interpretation and implementation of the standard.

Watching the simulations and observing waveforms is an excellent way to gain familiarity with the operation of the VC and its interfaces, whether standard or proprietary. Effective knowledge transfer is one of the most difficult problems for VC providers; the effort to document and deliver the testbench to the IP integrator may be more than offset if less support is required to help the users understand the VC.

In the case of a soft VC, the integrator may want to be able to rerun the standalone verification using the post-synthesis gate-level netlist in place of the original RTL model. This helps to check that the synthesis process didn't break the functionality of the VC, although it is a less comprehensive technique than equivalence checking for this purpose.

If the integrator has the rights and ability to modify the VC, most likely for a soft VC, then it may be possible to rerun the standalone verification environment to check that key features have not been broken by the changes made by the integrator. Of course, there are a number of caveats to this approach. If the changes are very extensive, the functionality may be different enough that the VC provider's standalone verification environment is no longer applicable. Many VC providers would argue that changing a VC is an inherently bad idea because it adds risk to the SoC project and places a support burden on providers; in fact, some VC licensing agreements explicitly prohibit any modification.

3.3 VC Re-Verification in an SoC

Ideally, the VC integrator would like to reuse as much of the standalone VC verification environment as possible during system-level SoC verification. In practice, the VC provider can offer some help but there are some significant limitations on verification reuse.

The biggest issue is that, after integration into the SoC, some or all of the VC I/O ports are "buried" within the chip. Chip-level and system-level verification strategies usually do not involve direct stimulation of internal signals. Therefore, the drivers and any portion of the testbench that drives VC inputs cannot be used directly in any non-standalone simulation. Similarly, the drivers usually can't check VC outputs because they only can determine correctness of results if they have control over the VC inputs.

If the VC is encapsulated by an internal scan chain, it may be possible to use the drivers to generate the serial data to apply to the VC inputs in a test mode and check the serial data captured from the VC outputs to determine correctness of results. This may be useful for manufacturing test purposes, but it really does not verify the VC any differently than does standalone simulation. More importantly, it does not check to see that the VC is integrated properly into the SoC.

If some of the VC inputs are connected directly to chip I/O pins, then it may be possible to use the drivers to stimulate at least these interfaces. For the buried inputs, if the drivers are written in a modular fashion (for example, using a set of tasks to generate stimuli and check results), then they may be able to serve as building blocks for portions of the SoC verification environment.

For example, a VC implementing a master-slave bus such as PCI may have these properties. The tests and drivers that access the VC (and SoC) as a slave may work perfectly well at the chip level. However, stimulating the VC as a master device requires modifying the drivers that accessed the now-buried VC inputs. The SoC verification environment must properly stimulate the chip input pins so that the logic inside the SoC stimulates the VC to initiate master cycles on the bus.

Fortunately, several parts of the standalone verification environment are inherently reusable at the chip level. All forms of monitors, including those for bus protocols and those derived from assertions, should only passively monitor the signals of the VC. Therefore, they should be easily reusable during SoC or system simulation since they can continue to monitor the same signals buried within the SoC. Depending upon the exact form of the monitor, the VC integrator might have to adjust hierarchical path references to properly point to the location of the VC inside the chip.

In addition to monitor reuse during simulation, assertions within the VC can be reused for model checking and other forms of formal verification. Finally, it is usually easy to repeat all forms of functional and code coverage while running the chip-level or system-level tests. This allows the VC integrator to compare the coverage results with those from VC standalone verification to determine whether the SoC is testing all required functionality of the VC.

4. Functional Verification Deliverables Rules

4.1 Drivers

4.1.1 Rules

VSI FV Rule 4.1.1: The testbench side of the driver interface must be documented.

Justification: It is required by the VC integrator in order to connect the driver correctly within the SoC verification environment.

VSI FV Rule 4.1.2: The driver should operate at the I/O boundary of the VC and not drive internal signals directly.

Justification: This allows the driver to be applied to different implementations of the VC, such as RTL and gate-level netlist.

4.1.2 Coding Guidelines

This section defines standard coding practice for VC drivers used in functional verification. These guidelines should be followed for any drivers used within the VC testbench, whether or not they are separated from the verification (such as testbench) code.

Drivers may respond to and drive the interface of the VC by accepting commands within the stimulus or events from monitors. The drivers may also be standalone elements. Standalone drivers are reusable without any dependencies on the testbench and stimulus control.

VSI FV Rule 4.1.3: A driver must not assign values to an interface signal more than once in the same time step.

Justification: This avoids glitches in simulation and aids in understanding the driver code.

VSI FV Rule 4.1.4: The VC must be driven only by self-contained drivers.

All stimulus interaction with the VC must come from common drivers that are modular and reusable.

Justification: This makes the VC verification environment more modular and reusable. Stimulus transactions work with an interface driver instead of just one VC.

VSI FV Rule 4.1.5: Each driver must stimulate and read only one interface.

The driver must stimulate and read only one interface into the VC. An interface is defined as a set of signals that implements a specific protocol.

Justification: This makes the design more modular and allows drivers to be reusable.

VSI FV Rule 4.1.6: Drivers must be self-contained.

Drivers must only be controlled by VC interface interaction, system stimulus, monitor output (directly or indirectly), and configuration code from the testbench. Drivers must not be dependent on other drivers or behavioral models.

Justification: Drivers must be standalone in order to be reusable as the block is migrated into a multi-unit or SoC environment. The designer must be able to change the internal portion of the design without the drivers changing.

VSI FV Rule 4.1.7: Drivers must drive all transactions that the interface can perform.

Justification: It must be possible to perform all interface activities by using the drivers. This enables development of stimulus that are based on transactions rather than timing. This also allows the capability to fully verify the interface and functionality of the VC.

VSI FV Rule 4.1.8: Drivers must have a procedural interface with input and output arguments.

Justification: Procedural interfaces are easier to encapsulate in modules and reuse.

VSI FV Rule 4.1.9: Global signals must not be used to configure drivers.

The use of global signals must be prohibited.

Justification: This enables the module to be portable.

VSI FV Rule 4.1.10: Drivers must not check the interface protocol.

Justification: Protocol checking must be handled by monitors.

VSI FV Rule 4.1.11: Inputs must be driven with legal values only for the duration that they are valid.

Justification: Leaving inputs at a constant value during invalid times does not detect cases where the VC under verification does not meet timing requirements. If a signal is not valid during a given time or cycle, it is preferable to drive that signal to an invalid value during this period, and ensure that the value does not propagate through the VC under verification causing failures. Three-state signals should be turned off during non-valid cycle times.

VSI FV Rule 4.1.12: Drivers should be partitioned for granularity of control.

It is recommended that all drivers and monitors be partitioned based on the granularity of control desired. Drivers should be partitioned with respect to the reuse of the driver at both the module level and the SoC level.

Justification: It is a good verification practice to avoid having one large “do-it-all” driver or monitor. Modularity also encourages reuse of driver and monitors during integration of a module at SoC level.

VSI FV Rule 4.1.13: Clocks should be used only to sample or produce data synchronously with a clock.

Justification: The verification environment should exercise the VC according to the interface protocol in the specification. If a clock is not specified, it should not be used to create and sample signals on the interface.

4.2 Monitors

4.2.1 Rules

VSI FV Rule 4.2.1: Monitors must be accompanied by appropriate debug information.

The debug information provided by the monitor to the user should be in a format that can be understood by the user. The debug information should translate the bug into a usage scenario, reference the appropriate section of the functional specification, or provide a contact number for the customer.

Justification: The VC integrator needs to be able to take appropriate action when an error is detected.

VSI FV Rule 4.2.2: The interface monitors should be split into two types: environment monitors and simulation-specific monitors.

- Environment monitors check that the testbench and the VC obey the environment constraints. For example, if the VC is an AHB slave, the environment monitors should check that the testbench generates valid AHB master inputs and that the VC responds according to AHB protocol.
- Simulation-specific monitors check that the testbench and the VC obey the specific requirements for that simulation. For example, consider a test where the VC is an AHB slave and the test generates user-mode accesses to addresses that must be accessed in supervisor mode. In this case, simulation-specific monitors should check that the testbench only generates user-mode accesses, and that the VC responds appropriately to such accesses by ignoring them or exhibiting the correct error behavior.

Justification: This allows a clear distinction between general-purpose protocol monitors that can be reused anywhere and monitors that are specific to certain tests.

4.2.2 Coding Guidelines

This section defines standard coding practice for VC monitors used in functional verification. These guidelines should be followed for any monitors used within the VC testbench, whether or not they are separated from the rest of the verification code.

VSI FV Rule 4.2.3: Monitors must monitor only one interface.

Justification: This makes the testbench more modular and reusable.

VSI FV Rule 4.2.4: Monitors must not drive design inputs.

Monitors must only listen to and monitor design and testbench interface signals. Monitors must also not drive internal signals.

Justification: Drivers and other testbench components (such as a clocking module) must do all signal driving to provide a consistent method for driving inputs.

VSI FV Rule 4.2.5: Monitors must check or observe all transactions on the interface.

Justification: The monitor must be able to check transactions generated by the driver or system-level test stimulus. This enables monitoring of all interface activity. This also enables development of stimulus that are based on transactions rather than timing.

VSI FV Rule 4.2.6: Monitors must verify the protocol on the external interface of a VC.

Monitors must verify the correct operation of the protocol for all legal operations on the external interface of a VC.

Justification: If a monitor relies on observing signals internal to the VC, there is a risk that these signals are either renamed or removed during device implementation, thus rendering the monitor ineffective.

Example: A monitor checks that an AMBA on-chip-bus protocol is adhered to by the VC, and therefore only monitors the signals of the VC at the AMBA bus level.

VSI FV Rule 4.2.7: Monitors must be self-contained.

Monitors must not rely on other code such as a driver, a behavioral model, configuration management software, the run-time environment, or configuration registers with the VC. If configuration parameters are required by the monitor, these must be passed via the monitor interface so it can run independently.

Justification: This makes the modules portable without reliance on other parts of the design, environment, or test code.

VSI FV Rule 4.2.8: Monitors must continuously monitor the interface.

Justification: Any gaps between invocations of the monitoring procedures may cause some output event to be missed.

VSI FV Rule 4.2.9: Monitors must not determine if a transaction should be in progress on an interface.

Monitors are intended only to check for correct operation of the transactions on the interface. They should not try to predict which operations should be taking place.

> **Justification**: Determining if a transaction should happen on an interface must be left to the test stimulus or block behavior checker.

VSI FV Rule 4.2.10: Monitors must only sample signals that will be preserved after synthesis.

Monitors must not reference internal signals that will not be present in all implementations; specifically signals in the RTL that may not exist in the gate-level netlist.

> **Justification**: If the monitor references internal signals that are not present after synthesis, the monitor is not reusable for gate-level simulations.

VSI FV Rule 4.2.11: Monitors must be reusable by all VCs that connect to the interface.

As the VC is integrated into a multi-unit or SoC environment, the monitors must be reusable as-is to check for violations on the interface to the VC.

> **Justification**: This allows portability of a monitor into the SoC environment and reduces design duplication. The system-level testbench inherits the entire interface checking of the lower level block, although this may require setting an HDL path to the VC. This helps to ensure that no system-level errors are introduced.

VSI FV Rule 4.2.12: Unrecognized interface behavior must be flagged as an error.

> **Justification**: This provides a capability to watch for any erroneous interface activity that violates a given interface protocol.

VSI FV Rule 4.2.13: Monitors must be capable of being established and disabled.

> **Justification**: This is important for reusing the VC on a SoC. On SoC designs, the pads often have more than one functionality (multiplexed IO), and the functionality is selected by primary pins or internal registers. Because of this, it must be possible to enable or disable the monitor according to the SoC setup.

VSI FV Rule 4.2.14: Each VC interface should have a single monitor.

The single monitor may encapsulate multiple modules or monitors as required. Internal drivers are removed as the block is integrated in an SoC environment but there should still be a single monitor for the interface. All checking at this point comes from the stimulus and original VC monitor.

> **Justification**: This makes reuse and instantiation at the SoC and system level easier.

VSI FV Rule 4.2.15: Monitor output should be kept to a minimum in the default configuration.

Too much debug information can make the log files difficult to read. Verboseness settings may be provided; the minimum output setting should be the default. Verboseness settings must be documented if used.

> **Justification**: This speeds up simulation and ease of debug.

VSI FV Rule 4.2.16: Monitors should provide abstractions of interface activity.

Monitors must be used to identify information on the interface and report bus activities to the testbench.

> **Justification**: The testbench or test stimulus may rely on an independent monitor to verify that a transaction has occurred. The abstraction may be used to coordinate and sequence downstream stimulus.

> **Example**: A bus monitor may report all stores to a certain address range that occur on the system bus.

4.3 Assertions

4.3.1 Rules

VSI FV Rule 4.3.1: Any assertions provided should be in specification form.

Justification: Providing the specification (source) for the assertions aids in the design knowledge transfer from the VC provider to the VC integrator, and allows use of the assertions in additional tools beyond those used in provider functional verification.

4.4 Functional Coverage

4.4.1 Input Data Functional Coverage

VSI FV Rule 4.4.1: The measurements of input data functional coverage must be documented in a manner that allows the VC integrator to assess the quality of the standalone VC verification.

Justification: In order to assess the extent to which the VC was verified, the VC integrator must review quantitative measures of its verification. Input data functional coverage quantifies the scope of the data stimulus applied to the VC during verification.

Example: This is a multi-dimensional organization example:

Dimension 1, instruction: **`add`**, **`sub`**, **`mul`**

Dimension 2, register: **`0`**, **`1`**, **`2`**, **`3`**

Full coverage space: instruction x register

Table 2: Example of VSI FV Rule 4.4.1

	add	**sub**	**mul**
register 0	25	56	21
register 1	13	16	93
register 2	0	72	33
register 3	45	0	41

4.4.2 Output Data Functional Coverage

VSI FV Rule 4.4.2: The measurements of output data functional coverage must be documented in a manner that allows the VC integrator to assess the quality of the standalone VC verification.

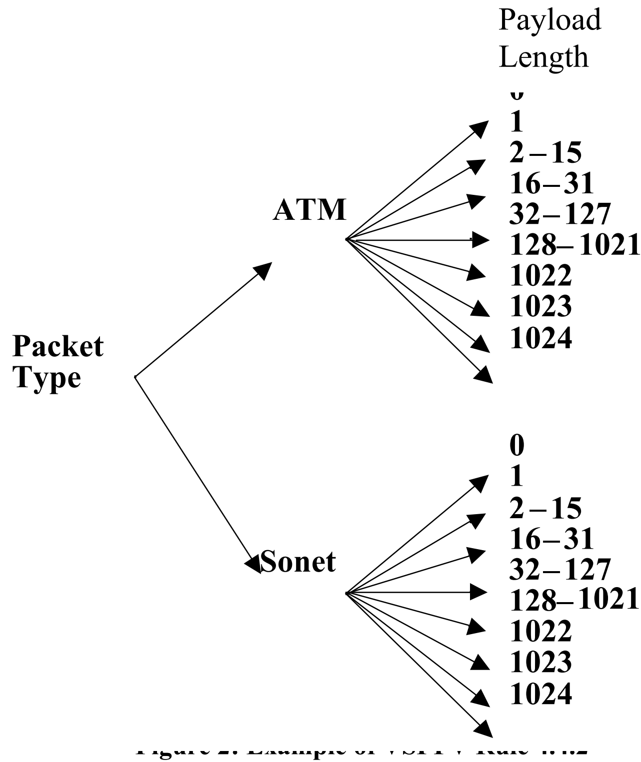
Justification: In order to assess the extent to which the VC was verified, the VC integrator must review quantitative measures of its verification. Output data functional coverage measurements quantify the scope of the observed response of the VC to the applied stimulus.

Example: A hybrid organization example follows.

Level 1, packet type: ATM, Sonet

Level 2, payload length: 0, 1, 2-15, 16-31, 32-127, 128-1021, 1022, 1023, 1024

Full coverage space: packet type, payload length



4.4.3 Internal Data Functional Coverage

VSI FV Rule 4.4.3: The measurements of internal data functional coverage must be documented in a manner that allows the VC integrator to assess the quality of the standalone VC verification.

Justification: In order for to assess the extent to which the VC was verified, the VC integrator must review quantitative measures of its verification. Internal data functional coverage measurements record the internal behavior of the VC in the data domain.

4.4.4 Input Temporal Functional Coverage

VSI FV Rule 4.4.4: The measurements of input temporal functional coverage must be documented in a manner that allows the VC integrator to assess the quality of the standalone VC verification.

Justification: In order to assess the extent to which the VC was verified, the VC integrator must review quantitative measures of its verification. Input temporal functional coverage quantifies the scope of the temporal stimulus applied to the VC during verification.

Example: This is a hierarchical organization example:

Level 1, request-to-grant interval: 1, 2, 3, >3

Level 2, grant-to-acknowledge interval: 1, 2, 3, 4-10, >10

Full coverage space: request-to-grant interval, grant-to-acknowledge interval

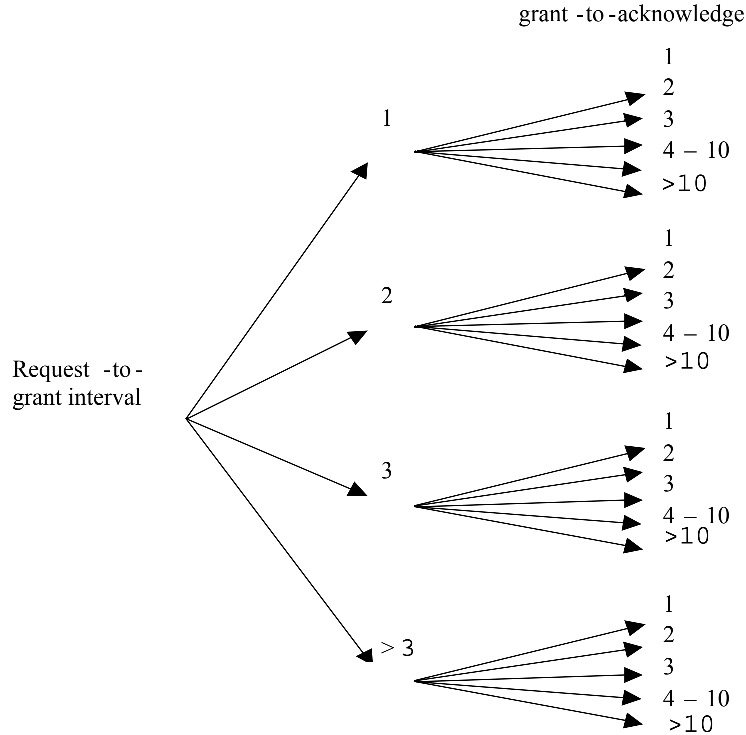


Figure 5: Example of VSI FV Rule 4.4.4

4.4.5 Output Temporal Functional Coverage

VSI FV Rule 4.4.5: The measurements of output temporal functional coverage must be documented in a manner that allows the VC integrator to assess the quality of the standalone VC verification.

Example: This is a multi-dimensional organization example:

Dimension 1, system 1 request type: read, write, read-modify-write

Dimension 2, system 2 request type: read, write, read-modify-write

Full coverage space: system 1 request type, system 2 request type

Table 3: Example of VSI FV Rule 4.4.5

	system 1 read	system 1 write	system 1 read-modify-write
system 2 read	25	56	**21**
system 2 write	13	16	93
system 2 read-modify-write	0	72	33

Justification: In order to assess the extent to which the VC was verified, the VC integrator must review quantitative measures of the VC verification. Output temporal functional coverage measurements quantify the scope of the temporal stimulus applied to the VC during verification.

4.4.6 Internal Temporal Functional Coverage

VSI FV Rule 4.4.6: The measurements of internal temporal functional coverage must be documented in a manner that allows the VC integrator to assess the quality of the standalone VC verification.

Justification: In order to assess the extent to which the VC was verified, the VC integrator must review quantitative measures of the VC verification. Internal temporal functional coverage measurements record the internal behavior of the VC in the time domain.

4.4.7 Input/Output Temporal Functional Coverage

VSI FV Rule 4.4.7: The measurements of input/output temporal functional coverage must be documented in a manner that allows the VC integrator to assess the quality of the standalone VC verification.

Justification: In order to assess the extent to which the VC was verified, the VC integrator must review quantitative measures of its verification. Input/output temporal functional coverage measurements record the input/output behavior of the VC in the time domain.

Example: This is a multi-dimensional organization example:

Dimension 1: SCSI bus phase: COMMAND, DATA, STATUS, MESSAGE IN,
SELECTION, RESELECTION
Dimension 2: Attention asserted: TRUE, FALSE

Table 4: Example of VSI FV Rule 4.4.7

	Attention Asserted	Attention Not Asserted
COMMAND	25	56
DATA	13	16
STATUS	60	72
MESSAGE IN	22	48
SELECTION	78	775
RESELECTION	34	2

4.5 Code Coverage

4.5.1 User-Defined Metrics for Code Coverage Measurement

VSI FV Rule 4.5.1: Only VC code must be instrumented for coverage.

The code instrumented for coverage must contain the entire module and exclude all verification code including the testbench, stimulus, drivers, monitors, models, and so on.

Justification: The module is the only part that is manufactured.

VSI FV Rule 4.5.2: Coverage should be run on all configurations that are manufactured.

Different configurations of a VC typically have significantly different code, with no single configuration including all possible code. In addition, different configurations typically have different features and will require different tests and possibly significant variations in the testbench.

Justification: Each configuration has different code and may require different tests, so the coverage results may be quite different.

4.5.2 Toggle Coverage

VSI FV Rule 4.5.3: The coverage reports must include the percentage of toggle (0-1 and 1-0) coverage achieved.

Justification: The VC provider must inform the VC integrator of the percentage of nodes that changed their polarity during the simulation verification. If the VC provider believes that 100% toggle coverage is not achievable, an exclusion file must be included, listing the conditions and reasons why certain nodes cannot be covered.

Example:

```
Toggle CountPosedge Count Negedge CountSignal
----------- ------------- -----------   -------
   5           6             5           ctl
   0           1             0           dval
   0           0             1           hold
   0           0             0           tmode
1889        1889          1888           xclk
2003        2002          2003           yclk
------------
3/6 covered = 50%
```

4.5.3 Statement Coverage

VSI FV Rule 4.5.4: The coverage reports must include the percentage of statement coverage achieved.

Justification: The VC provider must inform the VC integrator of the percentage of statements that were executed during simulation. If the VC provider believes that 100% statement coverage is not achievable, an exclusion file must be included, listing the conditions and reasons why certain statements cannot be executed.

Example:

```
Count   Line#   HDL Code
-----   -----   --------
   33      9      assign data = data_out;
          10
          11      always @ (posedge clk)
          12      begin
  127     13      state = next_state;
          14      end
-----
2/2 covered 100%
```

4.5.4 Branch Coverage

VSI FV Rule 4.5.5: The coverage reports must include the percentage of branch coverage achieved.

Justification: The VC provider must inform the VC integrator of the percentage of branches that were taken during simulation. If the VC provider believes that 100% branch coverage is not achievable, an exclusion file must be included, listing the conditions and reasons why certain branches cannot be taken.

Example:

```
Count   Line#     HDL Code
-----    -----     --------
   16    109      if (ack == 1)
    0    111      else

Count   Line#     HDL Code
-----   -----     --------
         115      case (state)
 0       116      2'b11 :
 232     118      2'b00 :
 252     123      2'b01 :
 16      128      2'b10 :
         134      endcase
-----
4/6 covered = 66.7%
```

4.5.5 Condition Coverage

VSI FV Rule 4.5.6: The coverage reports must include the percentage of condition coverage achieved.

Justification: The VC provider must inform the VC integrator of the percentage of sub-expressions that were tested during simulation. If the VC provider believes that 100% condition coverage is not achievable, an exclusion file must be included, listing the conditions and reasons why certain sub-expressions cannot be tested.

Example:

```
Line#   HDL Code
-----   --------
57      assign sig1 = ip1 && ip2;

Count   (ip1, ip2)
-----   ----------
5       0 0
2       0 1
0       1 0
3       1 1
-----
3/4 covered = 75%
```

4.5.6 Finite State Machine (FSM) Coverage

VSI FV Rule 4.5.7: The coverage reports must include the percentage of finite state machine coverage achieved.

Justification: The VC provider must inform the VC integrator of the extent to which the control structures in the RTL source code were exercised during simulation. If the VC provider believes that 100% FSM coverage is not achievable, an exclusion file must be included, listing the conditions and reasons why certain states, transitions and paths cannot be exercised.

Example:

```
State Coverage for   state:   3/4 (75.0 %)
States Covered          States Not Covered
--------------          ------------------
READ_INST               SPARE
READ_OP
WRITE_OP
Transition Coverage for state:   6/8 (75.0 %)

Transitions Covered     Transitions Not Covered
-------------------     ------------------
READ_INST->READ_INST    SPARE->READ_INST
READ_INST->READ_OP      WRITE_OP->WRITE_OP
READ_INST->WRITE_OP
READ_OP->READ_INST
READ_OP->READ_OP
WRITE_OP->READ_INST
```

4.6 Formal Methods

4.6.1 Verification Using Equivalence Checking

4.6.1.1 Rules

VSI FV Rule 4.6.1: The HDL provided for integration must be limited to the recognized synthesizable subset of the language.

In general, logic equivalence checking tools require synthesizable HDL as input.

Justification: This allows the integrator to verify different implementations of a VC using commercial equivalence checkers.

Example: A logical block coded in Verilog HDL must be limited to synthesizable constructs.

4.6.2 Equivalence Checking Data

VSI FV Rule 4.6.2: The equivalence checking data provided should include the following:

- Identification of any tools and versions used
- Any scripts
- Any logical constraints used
- All applicable name mapping rules
- Any other data needed to run logic equivalence checking between all views of the design at RTL level or below

Justification: This allows the integrator to easily reuse the same equivalence-checking methods or adapt for similar methods.

Example: Text files providing the recommended information, scripts, and guidelines.

4.6.3 Transistor or Gate-Level Equivalence Checking

VSI FV Rule 4.6.3: The transistor or gate-level equivalence checking data provided should include the following:

- Identification of any tools and versions used
- Any scripts
- Any other data needed to run logic equivalency checking between the RTL level design and the extracted version

Justification: This documents the verification done between the RTL and switch level.

Example: Text files providing the recommended information, scripts, and guidelines.

VSI FV Rule 4.6.4: All VCs containing a switch-level view and an RTL or gate-level view must include appropriate scripts and data for extracting an RTL or gate-level model from the switch.

Justification: This allows the integrator to compare an extracted design to the RTL or gate-level design using logic equivalence checking.

4.6.4 Dynamic Formal Methods

VSI FV Rule 4.6.5: The dynamic formal verification data provided should include the following:

- Identification of any tools and versions used
- List of assertions used as targets for the formal analysis
- List of constraints used for the formal analysis
- Identification of simulation tests amplified
- Inclusion of tests that were amplified as part of the VC verification environment

Justification: This allows the integrator to reproduce the dynamic formal analysis on the VC or potentially to reuse certain aspects of the documentation in static or dynamic formal analysis of the integration logic around the VC.

Example: Tests might be provided in C, C++, Verilog, VHDL, or a testbench language. The other deliverables are tied to a specific vendor tool and its proprietary formats.

4.6.5 Formal Coverage

VSI FV Rule 4.6.6: The formal coverage improvement data provided should include the following:

- Identification of any tools and versions used
- Description of the process used to improve coverage
- Inclusion of tests that were generated as part of the VC verification environment

Justification: This allows the integrator to assess how much coverage has been obtained through simulation tests and how much through the use of formal methods to improve the coverage from simulation. The generated tests should be provided as a way of performing regression testing on the VC.

Example: Tests might be provided in C, C++, Verilog, VHDL, or a test-bench language. The other deliverables are tied to a specific vendor tool and its proprietary formats.

4.6.6 Theorem Proving

VSI FV Rule 4.6.7: The theorem proving data provided should include the following:

- Identification of any tools and versions used
- Description of the specification model

- Description of the implementation model
- List of properties (relationships) proven

Justification: This allows the integrator to reproduce the theorem-proving models and theories, or to reuse proven theorems in hierarchical proofs relating to the integrated model.

Example: Theorem provers such as PVS or HOL may be used. The tool that is used determines the exact nature and syntax of the models and theories.

4.6.7 Model Checking and Property Checking

VSI FV Rule 4.6.8: The model checking or property checking data provided should include the following:

- Identification of any tools and versions used
- Environment or list of input constraints, as appropriate
- List of specification properties

VSI FV Rule 4.6.9: If model checking is used, an assume/guarantee style of model checking should be supported.

It is recommended that logical assertions be clearly defined describing the VC interfaces. These assertions represent the operating assumptions under which the VC was developed. By turning the assertions into the corresponding properties and verifying that the environment in which the VC is used meets them, the integrator ensures that the integrated system does not violate any operating assumptions on which the VC relies. This, in turn, ensures that the results previously verified for the VC alone continue to hold once it is integrated into a larger system. Furthermore, the results can be reused at higher levels of integration. These logical assertions may be implemented using a number of techniques, including model checking properties and checkers.

It is recommended that logical assertions be provided at the block level and below for any VC that may be modified or not integrated in its entirety.

Justification: This documents the design interfaces and allows the integrator to use a model checker to verify that the behavioral requirements of the interfaces are met. This prevents misinterpretation of interfacing rules.

Example: The following example uses the Open Verilog Library (OVL) to show a logical assertion for stability at the block interface. The Verilog version of OVL is shown. To verify that after an input signal *a_read* is asserted, the 8-bit input bus *abus* is not allowed to change value for two clock cycles:

```
assert_unchange #(0,8,2) int_abus_stable (clk, reset_n, a_read, abus);
```

VSI FV Rule 4.6.10: Model checking should be performed on control-intensive modules of a VC.

Justification: Model checking is most effective for control-intensive modules, where corner cases abound and state space explosion is generally limited. Model checking tends to be less effective for memory-intensive modules, due to state space explosion.

Example: Model checking might be performed on a cache controller, for example, to ensure that the cache coherency algorithm cannot fail.

4.6.8 Formal Constraint-Driven Stimulus Generation

VSI FV Rule 4.6.11: If formal methods are used to generate simulation stimulus, the following data should be provided to the integrator.

- Identification of all tools and versions used
- The set of constraints under which the stimulus have been generated
- Inclusion of tests that were generated as part of the VC verification environment

Justification: This allows the integrator to assess how many of the verification tests have been obtained through formal, constraint-driven methods as opposed to traditional methods of hand-written directed tests and random test generation. Documenting the set of constraints indicates to the VC integrator which portion of the input space is explored by the generated test cases. The constraints may or may not represent restrictions on the environment under which the VC operates. For example, constraints can be set to target specific corner cases. The generated tests should be provided as a way of performing regression testing on the VC.

Example: Tests might be provided in C, C++, Verilog, VHDL, or a test-bench language. The other deliverables are tied to a specific vendor tool and its proprietary formats.

4.6.9 Symbolic Simulation

VSI FV Rule 4.6.12: The symbolic simulation data provided should include the following:

- Identification of any tools and versions used
- The set of symbols used
- The tests or testbenches used

Justification: This allows the integrator to apply the pre-existing symbols if symbolic simulation is reused.

4.7 Documentation

4.7.1 Provider Functional Verification Documentation

4.7.1.1 Compliance Tests Carried Out

VSI FV Rule 4.7.1: The provider functional verification documentation must document all third-party test suites used to determine compliance to a standard.

Justification: A third-party test suite can provide a known reference point against which the VC can be judged. Note that the third-party material may have to be licensed separately in order for the VC integrator to reproduce the tests.

Example: VC providers might use an AMBA Verification environment, provided by ARM, Ltd., to prove that their VC complies with the AMBA bus-interface specification.

Example: A VC provider might use a third-party verification suite to prove compliance to the USB standard.

VSI FV Rule 4.7.2: The provider functional verification documentation must document all standard test suites used.

Justification: Where relevant industry standard tests exist and have been used to augment other functional tests, these must be documented, since they are useful references.

Example: Using the CITT G726 ADPCM Codec Test Patterns to validate an ADPCM codec.

VSI FV Rule 4.7.3: The provider functional verification documentation must document all hardware test suites used.

Justification: Although hardware implementation is typically not a deliverable, the fact that the VC has been successfully validated in a hardware platform is of value. Hardware validation takes place at or near real execution speed, and allows more functionality to be checked. All the functions verified in the validation should be documented or included in the functional coverage document.

Example: A hardware implementation that has been used to prove compliance with a standard, for instance an FPGA implementation of a USB VC that has been successfully validated at a USB "Plug Fest."

4.7.1.2 Functional Verification Environment—Tool and Platform Configuration

VSI FV Rule 4.7.4: The provider functional verification documentation must document the tool and platform configuration used.

Justification: Although quality design and verification deliverables should not be dependent on features of a version of a software tool or platform, they sometimes are. As part of the overall configuration control for the VC, the VC provider should document what software tools and tool versions were run on which platform to verify the VC. This enables providers or users to reproduce the environment. It also allows users to determine if there are risks associated with their chip-level verification tools environment-for instance, by using a different simulator than the provider used.

Example: "Version 2.3 of the M16550A UART was verified with V5.0 of the testbench, using V4.5 of Verilog-XL and V5.6 of ModelSim on a Solaris 2.8 machine."

4.7.1.3 Functional Verification Environment—Details

VSI FV Rule 4.7.5: The provider functional verification documentation must document the details of the environment used for VC verification.

Justification: This document gives an end user an insight into the architecture of the functional verification environment and the purpose of each of the functional models used.

Example: A short description of the environment that includes a block diagram showing how the different parts of the environment interact with each other, an overview of the function of each of the functional models used, and how the tests are controlled.

VSI FV Rule 4.7.6: The provider functional verification documentation must document the design languages used (for example, VHDL or Verilog).

Justification: This allows the end user to determine any potential mismatch between the simulation environment and that originally used to verify the VC. For instance, the VC integrator might have only Verilog simulation tools available, yet the VC might have a VHDL testbench.

Example: "The MCAN VC was implemented in both VHDL and Verilog, but verified using a VHDL testbench containing VHDL functional models."

VSI FV Rule 4.7.7: The provider functional verification documentation must document the directory structure used.

Justification: This type of information allows rapid comprehension of the functional verification environment.

Example: A directory tree diagram illustrating the location of the different verification environment files.

VSI FV Rule 4.7.8: The provider functional verification documentation must document the testbench files and structure used.

Justification: This allows the end user and the maintainer to understand how the different components of the testbench fit together, and how any files associated with the testbench relate to those components.

Example: A block diagram of the testbench showing how the different parts of the testbench connect to each other.

VSI FV Rule 4.7.9: The provider functional verification documentation must document the models (memory or other functions) used.

Justification: Models are a vital part of the functional verification environment. The documentation allows the end user to understand the function of the models, the way in which they interface to the rest of the environment, and how their behavior can be controlled. Limitations in the accuracy or fidelity of the model should be listed.

Example: A datasheet for each of the models used.

VSI FV Rule 4.7.10: The provider functional verification documentation must document the stimulus generator (including drivers) used.

Justification: Stimulus generation is a major part of standalone verification. If a stimulus generator is used, its functionality, interface, and control methods should be documented so that functional test events can be debugged back to their root cause.

Example: A datasheet for each of the stimulus generators used in the environment.

VSI FV Rule 4.7.11: The provider functional verification documentation must document the response checker (including monitors) used.

Justification: VC functional test environments must be self-checking in order to facilitate the assembly of regression tests. The functionality of the response checker can vary from a check of actual results against a golden result file to a temporal model that checks that events occur within a certain simulation time window. The response checker's functionality should be documented, including a description of any driver files it may require, and a method for determining that the response check has passed.

Example: A datasheet for each response checker used in the environment.

VSI FV Rule 4.7.12: The provider functional verification documentation must document the bus functional model (BFM) for on-chip bus (OCB) VCs.

Justification: Bus functional models are used to provide simplified bus agent models for verifying on-chip bus VCs. The behavior of the BFM should be documented, along with information on the structure and location of control files for the model. The limitations of the BFM should be noted.

Example: A datasheet for each BFM used in the environment.

VSI FV Rule 4.7.13: The provider functional verification documentation must document all the software drivers used.

Justification: In a functional verification environment that enables both hardware and software to run together, software drivers may be used to implement the low-level, hardware-software interface. In this situation, the software drivers used should be documented, along with their versions. Ideally, these drivers are also available as separate deliverables for the VC.

Example: This might range from a reference to the software driver documentation, to a full description of the software driver including details of its API.

VSI FV Rule 4.7.14: The provider functional verification documentation must provide user documentation.

Justification: This information is useful for reproducing the functional verification environment and for rerunning the test suite without reverse-engineering the files.

Example: The VC provider must document the scripts used to compile the verification environment and to run the functional verification suite.

VSI FV Rule 4.7.15: The provider functional verification documentation must document the design parameterization options and how they were checked.

Justification: In the case where a VC has configuration parameters that affect the structure and functionality of the design, extra care has to be taken to ensure that a range of values have been tested, and that the range chosen tests any potential functional problems. In turn, this implies that the verification environment is also parameterizable. The method by which the environment is parameterizable should be documented, along with the parameter values actually tested.

4.7.2 Functional Verification Deliverable Documentation

4.7.2.1 Directory Structure and Setup

VSI FV Rule 4.7.16: The functional verification deliverable documentation must document the directory structure used and setup.

Justification: The information provided helps set up and debug the user's environment.

Example: This is only an example, and differs for each VC provider.

- `run_vc` or `run_periph/`
  This is where simulation run scripts for VC and peripheral tests and associated files are located.
- `synth/design_compiler/`
  This is where the synthesis scripts are located.
- `synth/primetime/`
  This is where the static timing analysis scripts are located.
- `golden_src/`
  This is where the original VC and peripheral source (RTL code) files are located.
- `vc_setup`
  This details the configuration of source files that a user must execute to set up the VC environment.

VSI FV Rule 4.7.17: The functional verification deliverable documentation must provide a manifest of all the deliverables and a description of the deliverables, file names (meaningful names), size (in KB), and so on.

Justification: A list of deliverables (documentation and design collateral) provided for the VC helps in understanding and integrating the VC in a user's environment.

Example: Listed below are some of the deliverables for a soft or hard VC. This varies for different VC providers.

- Datasheet: should contain the name of the VC, key features and limitations, VC description, system block diagram, pinout and description, and performance specs and data
- VC Architecture specification
- Source code: actual VHDL or Verilog code
- VC environment, tools and setup: lists all the project environment setups, files required, tools and version numbers, and so on
- Test vectors: actual test vectors used in verification, and format (serial, parallel)
- Testbench, checkers, BFM information

VSI FV Rule 4.7.18: The functional verification deliverable documentation must document how to compile the verification environment.

Justification: VC providers should document the procedure, tools, and scripts required to compile the verification environment so that users can reproduce their results.

Example: Listed below is an example of what should be included in the document and how to perform the above tasks. This varies for different VC providers.

- Execute any VC provider setup scripts to set the environment variables and the tools needed for verification checks.
- Identify all the component parts that are required to compile the VC functional verification environment (for example, testbench files, models, stimulus and response files, command files, and so on).
- Use make scripts or any other scripts or commands to compile the verification environment.

VSI FV Rule 4.7.19: The functional verification deliverable documentation must document how to use, set up, and debug the testbench.

Justification: The VC provider should document how to use, set up, and debug the testbench provided, to ensure that the VC provided works and can be easily integrated into the end user's environment.

Example: The document should include (but not be limited to) “How to Modify” and “What to Modify” information related to user environment setups, debug of tests, and testbench. The extent to which documentation is provided is dependent on the complexity of the VC.

VSI FV Rule 4.7.20: The functional verification deliverable documentation must provide the verification steps to be performed by the VC integrator.

Justification: The purpose of the verification steps is to ensure that there are no missing links (such as environment variable dependencies or pointers to data), and that all tests work at the module or full chip level. The steps are different, depending on whether the VC is soft, hard, or firm.

Example: The following example shows the verification steps for soft or hard VCs. This varies for different VC providers.

- Soft modules: For soft modules, the recommended steps might be to compile (RTL code), do some sort of automated rules or guidelines checks, simulate the design, synthesize, and do some type of formal verification.
- Hard modules For hard modules, the recommended steps might be to compile (RTL code if provided), do some sort of automated rules or guidelines checks, generate top level or individual netlist, and perform design rule and schematics checks for the design.

VSI FV Rule 4.7.21: The functional verification deliverable documentation must document how to interpret the results of the deliverable tests.

Justification: The VC provider should document the minimum requirements to check environment functionality, and how to check the results (error interpretation or description of pass or fail decision) of all the delivered functional tests.

Example: While running regression tests or multiple different tests, diagnostic information should be provided on whether a test performed according to the specification. For example, if a test failed, information on why the test failed should be provided and displayed; if a test ran successfully with no errors, “Test result: Pass” should be displayed.

VSI FV Rule 4.7.22: The functional verification deliverable documentation must document how certain deliverables could be reused for system-level verification.

Justification: The VC provider should document which (if any) of the deliverables (tests, testbench, scripts) provided can be run for system-level verification checks.

Example: If the testbench or tests are designed to work at the unit level, is it possible to use the same testbench (with or without minor modifications) at system level testing? The tests provided can be written such that they can be used at unit- and system-level verification. These are only a few examples stated. This rule is not limited to tests and testbench.

VSI FV Rule 4.7.23: The functional verification deliverable documentation must document any tests that cannot be reproduced using the functional verification deliverables.

> **Justification**: VC providers do not necessarily deliver their complete regression suite to the end user, for a variety of practical and commercial reasons. However, the end user should know which of the provider tests cannot be reproduced by the functional verification deliverables.

4.8 Behavioral Models

VSI FV Rule 4.8.1: The detailed behavioral model must be configured independently of the VC.

Block behavior checkers must be configured using the testbench, stimulus, or monitors. Configuration of the behavioral model must not occur based on the internal VC state. Behavioral code may be used to preload states into the VC.

> **Justification**: Problems with the VC configuration may be masked. This provides independent verification of the VC operation and configuration mechanism.

4.9 Scripts

4.9.1 Simulation (Verification) Scripts

4.9.1.1 Environment (Types and Partitioning) Scripts

VSI FV Rule 4.9.1: The regression environment must support running stimulus with a single submission.

The regression environment must support running all patterns and allow the generation and re-simulation of stimulus.

> **Justification**: This makes running regression tests a push-button routine. In addition, a user may only want to run a particular set of patterns.

VSI FV Rule 4.9.2: The regression log file should contain all information needed to reproduce the run.

> **Justification**: The VC integrator needs to be able to re-create original verification conditions and confirm correct operation of VC and the verification environment.

VSI FV Rule 4.9.3: Simulation output files must be named consistently across simulation environments.

The following files must be provided, and the following file extensions are recommended:

- .log

  Messages generated by the regression stimulus displays, drivers, and monitor reporting
- .sum

  A summary log file containing results of the simulation (that is, pass or fail) and any stderr from script or simulator invocations (for example, core dumps, license unavailable, and so on).

> **Justification**: This allows regression environments to search for results, errors, and any other necessary files based on these file extensions. Integrators can "grep" a log file to make sure the simulator and testbench did not produce any errors.

VSI FV Rule 4.9.4: The testbench and a subset of stimulus must operate on any delivered gate-level models.

> **Justification**: The VC integrator must be able to prove stimulus runs on low-level models even if formal equivalency checking and static timing analysis is used. This is used to prove that the design is implementable.

VSI FV Rule 4.9.5: Hard VC models must operate with back annotation.

The regression environment must allow SDF regressions to be run. The ability to vary the SDF back-annotations for various corner simulations during the regressions must be supported.

Justification: This allows VC or SoC timing to be verified using patterns in an automated manner, and guarantees quality and reuse of testbench components and library.

VSI FV Rule 4.9.6: It is recommended to run zero- and unit-delay gate-level regressions.

Justification: This guarantees quality and reuse of testbench components, libraries, and stimulus. It also ensures that the design is implementable.

4.9.1.2 Regression (Job Control) Simulation Scripts

VSI FV Rule 4.9.7: The verification environment must be re-creatable.

Justification: The VC integrator must be able to recreate the simulation. This allows proof of the verification claims.

Example: Provide tool version numbers or identification criteria, OS patches needed, makefiles, startup scripts, environment variable settings, defines, and ReadMe files that describe the work flow.

VSI FV Rule 4.9.8: Every regression test must be able to be run standalone.

Justification: Running the complete regression test suite may take some time, which is not desirable, if only one of the regression tests is failing and multiple runs of this regression test are needed to determine the cause of this failure. The ability to run individual tests from the regression suite avoids such situations.

VSI FV Rule 4.9.9: Regression tests must not rely on the results of a former regression test run.

Justification: Dependencies between regression tests are often hard to identify, and can cause unreliable results. If a regression test must rely on the result of a previous run, it is better to provide a single regression test with multiple steps.

4.10 Stub Model

VSI FV Rule 4.10.1: A stub model should be provided in the same format as the VC itself.

Justification: This allows the stub model and the full VC to be interchangeable in the same testbench.

4.11 Functional Verification Certificate

VSI FV Rule 4.11.1: The functional verification certificate must consist of a header and a body. The header must contain the information listed in the following format:

```
VSIA Functional Verification Certificate Format - V1.0
VC_name: <name>
VC_version: <version>
Env_name: <name>
Env_version: <version>
Date: <date>
Environment Configuration: <configuration>
VC Parameter Configuration: <parameter_configuration>
```

Example:

```
VSIA Functional Verification Certificate Format - V1.0
VC name: My_virtual_component
VC Version: Version 2.5
Env name: My_verification environment
Env version: Version 1.8
Date: Fri Jan 11 20:42:34 MST 2002
Environment Configuration: Solaris 2.8
                                        ModelSim 5.5e
                                        Perl5.1
VC Parameter Configuration: BusWidth = 16, NoInts = 7, DMAController=0, Speed=15
```

Justification: A standard format for the text of the certificate allows for mechanical comparison of the provider-generated certificate and the user-generated certificate.

VSI FV Rule 4.11.2: The body of the functional verification certificate must consist of multiple lines naming the test run and the outcome in the following format:

```
Run_<name>: Outcome
```

Justification: One-to-one mapping of test name and outcome allows users to easily identify tests that failed for further investigation.

VSI FV Rule 4.11.3: Outcomes included in the functional verification certificate must be limited to "passed" or "failed."

Example:

```
Run_USB_Chapter9: passed
Run_USB_Suspend: passed
Run_USB_High_Speed_with_data_errors: failed
Run_USB_Reset: passed
```

Justification: The purpose of the functional verification certificate is to give a high-level check on the verification of the VC; therefore, only binary outcome values are required. Further detailed debug messages are available in tool transcripts or log files.

VSI FV Rule 4.11.4: The functional verification certificate must be accompanied by a documented procedure to generate the certificate.

Example: The VC provider includes a script that runs all tests included in the verification certificate.

Justification: This eliminates the need for trial and error in the generation of the verification certificate.

5. Testbench Coding Guidelines

5.1 Coding for Verification

This section contains the rules and guidelines for verification specific coding. The rules should form a subset of the VC supplier's design coding standards and could be integrated into their coding standard.

There has been a special effort to make as many of the standards as language- and tool-independent as possible. Standard coding guidelines aid VC creators and integrators to share design data and develop SoC solutions. The following rules and guidelines apply primarily to the Verilog HDL. Similar constructs or methods must be used for other languages, if applicable.

5.2 General

This section contains general rules and guidelines for coding verification components.

VSI FV Rule 5.2.1: The latest version of the VC supplier's HDL coding standard must be adhered to.

Justification: The verification HDL coding standards are written to be a super-set of the HDL coding guidelines.

Example: The rules and guidelines for coding SRS-compliant HDL are excerpted from the Motorola SRS document and are provided in the Verilog HDL section.

VSI FV Rule 5.2.2: Synthesizable and behavioral verification code must be partitioned separately.

Justification: This eases mapping to emulation and hardware accelerators.

VSI FV Rule 5.2.3: Comments must describe the intent and purpose of the code.

Justification: The intent and purpose of the code needs to be conveyed, not simply a description of what the code does. Therefore, explaining the functionality and not the implementation should be the primary focus.

Example: *bad*

```
// Increment counter
       rx_addr_ptr = rx_addr_ptr + 1;
```

Example: *good*

```
// Increment address pointing to next available location in // receive buffer

       rx_addr_ptr = rx_addr_ptr + 1;
```

Example: *bad*

```
--Increment counter
       rx_addr_ptr <= rx_addr_ptr + x.01;
```

Example: *good*

```
--Increment address pointing to next available location in

--receive buffer
       rx_addr_ptr <= rx_addr_ptr + x.01;
```

VSI FV Rule 5.2.4: If three-state buffers are not desired, the value 1'bz must not be used.

Justification: This avoids unintended three-state buffers.

VSI FV Rule 5.2.5: Unless variables are used globally, local declarations must be in named blocks.

Justification: This avoids accidental interferences with other blocks that may produce unexpected results.

5.3 Symbolic Constants

This section contains rules and guidelines for symbolic constants. In Verilog, symbolic constants are text macros and parameters. In C++, they are typed constants.

In Verilog, defines and parameters have advantages and disadvantages. This should be taken into account when deciding which one to use. For example, parameters allow different values to be assigned to different instances. Defines allow setting values for multiple modules from a single location.

Where both parameters and defines do the job equally well, parameters are preferable because they eliminate the problem of a conflict with using the same name defined somewhere else.

VSI FV Rule 5.3.1: Address offsets should be specified by defines and named consistently.

The offset to a base address should be specified with a define, for example, as `<module_prefix>_<reg_name>_OFFSET`.

Justification: Changes in register maps are easier to handle. It also makes the code more readable, and allows for consistency and ease of relocating devices and registers in memory.

VSI FV Rule 5.3.2: Base address names should be specified by defines and named consistently.

Justification: This allows for consistency and ease of relocating devices in memory.

Example: `<module_name>_BASE = 100`

VSI FV Rule 5.3.3: Defines should be used for frequently used values.

If a value is used frequently in a specific standalone device stimulus, use a define statement.

Justification: Defines improve readability and simplify the porting of a stimulus to different environments and chips.

VSI FV Rule 5.3.4: Text macros should be centralized in one or more file locations.

Justification: Text macros may have global scope. If they are scattered, it becomes more difficult to manage the data.

VSI FV Rule 5.3.5: It must be possible to set parameters from the simulator command line.

This allows configurations to be managed external to the testbench.

Justification: The VC does not have to be modified to run a new configuration.

VSI FV Rule 5.3.6: The default parameter settings must specify a verified implementation.

Justification: This ensures that the testbench operates in the default configuration.

5.4 Routines

Efficient verification stimulus generation relies on using routines, macros, functions, and so on, to raise coding to a higher level of abstraction. This technique helps to avoid coding errors caused by cut and paste of common code blocks. It is useful to standardize the format of routine names to ease identification and classification of routine calls, and to ensure uniqueness in an SoC environment.

VSI FV Rule 5.4.1: All module-specific routine names must be easily identifiable.

Routines that are only used by one module must be named consistently, for example, in lower case.

Justification: Improves readability and makes it easy to determine if a routine call is module specific (local).

VSI FV Rule 5.4.2: All module-specific routines must be made unique.

All module-specific routines should be named to ensure that they are unique across the whole design. For example, every such routine could be preceded by at least two characters unique to the module. Failure to precede new routines with the unique characters could produce warning or error messages and the incorrect redefinition of a routine. Module-specific routines imply routines that are not part of the common command set.

Justification: Many modules share similar routine names. Using a generic routine name may cause non-execution of the simulation.

Example: `<module_name>_try_routines` instead of `try_routines`

VSI FV Rule 5.4.3: Routines should be disabled internally.

Routines should not be disabled from other parts of the verification code if they can be disabled internally. If disabling code is required, a named begin/end sequence should be disabled from within the task. Disabling routines in this way is a feature of Verilog. There is no simple equivalent construct in VHDL.

Justification: This avoids unspecified simulator behavior.

Example:

```
task write
begin: write_access
    if (abort) begin
       disable write_access;
    end
  ...
end
```

VSI FV Rule 5.4.4: Routines should be disabled from a single location.

If routines cannot be disabled internally, it is recommended to have all the disable constructs originate from the same block of code.

Justification: This confines maintenance to a single location.

5.5 Signal State and Time

Using internal signals for verification is discouraged. Internal signal names may change or the signal may be removed during the downstream design implementation process. In this case, considerable rework of the verification code may be required. When supplying a VC to integrators, it is unreasonable to place them in the situation where they need to modify the verification code and where they may fail to detect an implementation error because of an error in making the changes.

If internal signals are used for verification, care must be taken to ensure that problems do not arise during mixed-mode simulation and production verification. In addition, the insertion of delays must be based upon module clocks. If fixed delays must be used, they must be parameterized for compatibility with different simulators and testbenches.

VSI FV Rule 5.5.1: Internal signals referenced by stimulus must be listed in a single location.

Reference to internal signals is strongly discouraged, but is not prohibited. For the rare situations in which it is justified, it must be done using a consistent method.

Justification: This makes it easy to locate any internal signals used.

VSI FV Rule 5.5.2: Reference to internal signals must be by using text macros.

Reference to internal signals is strongly discouraged, but is not prohibited. For the rare situations in which it is justified, it must be done using a consistent method.

> **Justification**: Internal signals may not survive synthesis. They may be removed entirely, or exist by a different name. This rule provides for name change of signals referenced by the stimulus.

VSI FV Rule 5.5.3: Internal signals referenced by stimulus must be preserved through synthesis.

Appropriate synthesis constraints must be applied to guarantee that signals referenced by tests are not be deleted during synthesis.

> **Justification**: Internal signals may not survive synthesis. They may be removed entirely, or exist by a different name. This rule prevents signals referenced by tests from being deleted.

VSI FV Rule 5.5.4: Signals should be referenced at the module boundary.

Signals referenced by simulation should cross a module boundary. The boundary may not be flattened during synthesis (preserved).

> **Justification**: This is a secure way to preserve the signal names between RTL and gate level netlist.

VSI FV Rule 5.5.5: Internal signals must not be used to determine pass or fail of a test.

Only observed signals on the VC boundary can be used to determine pass or fail. This is so the tests can be run on a tester. This does not preclude using such signals for providing additional debug information.

If the VC is more complex, signals may be brought through an additional test bus to observe required internal points (state vectors, and so on) to determine pass or fail. The test bus crosses the VC boundary. It is also possible to observe this test bus in a special test mode on a full chip simulation on the pads. Because of this, it is possible to use this bus for silicon debugging.

VSI FV Rule 5.5.6: Internal signals must not be forced during verification.

It is recommended to only drive signals on the VC boundary to verify the VC. Internal signals are not visible on the tester; therefore, any forces have no meaning or effect.

> **Justification**: Internal forces cannot be implemented on a tester. So the device must operate in simulation without any internal forces and use special test modes instead of the forces.

Exception: Forcing signals to test isolation of power supply modes.

5.6 Clocking

A difference in clocking techniques is often a source of incompatibility when integrating modules into a SoC environment. Multiple clock environments and clock skew must be handled consistently. Special clocking may also be needed to speed up simulation and address production test needs. The following clocking rules are HDL design constraints, and will move to the HDL coding standards, if appropriate.

VSI FV Rule 5.6.1: Clocks must be scalable using a constant or variable.

Multiple clock environments must have a mechanism to allow period scaling of all clocks through a common constant or variable.

> **Justification**: This allows for quick portability between designs or versions. It allows adjustments for unit-delay simulations or test-bench clocks to match other environments with respect to the clocks and non-clocked signals.

If a unit delay is applied to every gate output during simulation, the clock width might be too narrow to accommodate the maximum gate depth. Rescaling allows this simulation to execute.

VSI FV Rule 5.6.2: Hard time delays must be parameterized.

Delay parameters must be specified as a fraction of the system level clock.

Justification: This allows time delay to be adapted to new simulation clocks without changing the stimulus code.

VSI FV Rule 5.6.3: Clock expressions must not round or truncate.

When using expressions to compute delay times, it must be assured that the expression operation does not truncate the fractional part of the delay. Using the expression "clock/2" may result in rounding errors if timescale and precision are not selected appropriately.

Justification: In order to precisely place edges in simulation, proper timescale and precision must be chosen.

VSI FV Rule 5.6.4: Non-derived clocks must use explicit assignments [0,1].

Justification: Use of non-explicit assignments could result in issues due to the dependence on the initialization of the clock.

Example: Using assignments of 1 or 0 instead of clk and ~clk.

VSI FV Rule 5.6.5: Derived clocks must be generated within the same simulator time step.

Justification: Use of derived signals may cause unintended skew between the base signal and derived signal. Generate all derived clocks in the same process block as the source call to avoid zero time-evaluation skew between clock edges.

VSI FV Rule 5.6.6: Clocks must not be initialized in initial blocks (Verilog only).

Justification: If a clock is initialized in an initial block and generated in an always block, there is no guarantee of sequencing. The always block may execute prior to the initial block.

VSI FV Rule 5.6.7: Master system clock phase width should be identified with a define using an obvious name.

It is recommended to use an easily identifiable name for the constant or variable that determines the master system clock phase width into an SoC (for example, MPHASE).

Justification: This ensures readability and portability between system environments.

Example: The Verilog example for master clock generation at the testbench level is:

```
'define MPHASE 100
always
       begin
              #('MPHASE);
              CLK = 'HIGH;
              #('MPHASE);
              CLK = 'LOW;
       end
```

Example: The equivalent VHDL example is:

```
CONSTANT MPHASE : time := 10 ns;
CONSTANT HIGH   : std_logic := '1';
CONSTANT LOW    : std_logic := '0';
PROCESS
BEGIN
    WAIT FOR MPHASE;
    clk <= HIGH;
    WAIT FOR MPHASE;
    clk <= LOW;
END PROCESS;
```

Example: The equivalent *e* example is:

```
define MPHASE 100;
event clk is {@clk; delay(MPHASE)} @sim;
on clk {
    'CLK' = 1 - 'CLK'
}
```

VSI FV Rule 5.6.8: Phase widths of non-system clock inputs should be identified using an obvious name.

For example, the phase widths of non-system clock inputs should be named *<module_name>_<clock_pin_name>_PHASE*

Justification: This ensures readability and portability between system environments.

Example: The Verilog example for a SPI clock is:

```
'define SPI_SCLK_PHASE 'MPHASE/4
always
       begin
              #('SPI_SCLK_PHASE);
              SCLK = 'HIGH;
              #('SPI_SCLK_PHASE);
              SCLK = 'LOW;
       End
```

Example: The equivalent VHDL example is:

```
CONSTANT SPI_SCLK_PHASE : time := MPHASE/4;
PROCESS
BEGIN
  WAIT FOR SPI_SCLK_PHASE;
  sclk <= HIGH;
  WAIT FOR SPI_SCLK_PHASE;
  sclk <= LOW;
END PROCESS;
```

VSI FV Rule 5.6.9: Hard-coded delays should be used only to create clock skew.

It is recommended that standard task calls be used to count the number of clock cycles before the next desired task starts.

Justification: Hard-coded delays create erroneous results, since frequencies and module access times change between different environments and chips.

Exception: Analog circuits that require delays for modeling, such as phase lock loops.

VSI FV Rule 5.6.10: Skew must be placed on clock edges.

Multiple clock environments that require synchronization between clock domains must place skew between clock edges for event-based simulations.

Justification: This avoids edge-on-edge issues in simulation that might produce incorrect results. Synchronizing latches between clock domains potentially slips or adds a clock of delay to the output if no skew exists. This helps event-based simulations overcome simulator deficiencies and discrepancies. Event queues of the simulators may be different with blocks simulated in the SoC environment instead of the VC environment.

VSI FV Rule 5.6.11: Skewed clocks must be named consistently.

Skew between clocks must be represented in the naming convention. For example, use *P#* for positive skew and *N#* for negative skew for event-based simulations.

Justification: This ensures readability and portability between system environments.

Example: The Verilog example for an SPI clock with skew to avoid synchronization issues is:

```
'define SPI_SCLK_PHASE 'MPHASE/4
'define SPI_SCLK_PHASE_P1 'SPI_SCLK_PHASE + 1
'define SPI_SCLK_PHASE_N1 'SPI_SCLK_PHASE - 1

always
       begin
              #('SPI_SCLK_PHASE_P1);
              SCLK = 'HIGH;
              #('SPI_SCLK_PHASE_N1);
              SCLK = 'LOW;
       end
```

Example: The equivalent VHDL example is:

```
CONSTANT SPI_SCLK_PHASE : time := MPHASE/4
CONSTANT SPI_SCLK_PHASE_P1 : time := SPI_SCLK_PHASE + 1
CONSTANT SPI_SCLK_PHASE_N1 : time := SPI_SCLK_PHASE - 1

PROCESS
BEGIN
  WAIT FOR SPI_SCLK_PHASE_P1;
  sclk <= HIGH;
  WAIT FOR SPI_SCLK_PHASE_N1;
  sclk <= LOW;
END PROCESS;
```

Example: The equivalent *e* example is:

```
clock_driver()@sys.any is {
              while TRUE do {
                 delay(SPI_SCLK_PHASE_P1);
                 'SCLK' = HIGH;
                 delay(SPI_SCLK_PHASE_N1);
                 'SCLK' = HIGH
              }
          }
```

VSI FV Rule 5.6.12: Clock min and max skew should be modeled.

Justification: Clocks should have min and max skew modeled to expose the timing relationship between clock domains and sequencing of testbench components.

VSI FV Rule 5.6.13: Clock inputs should be able to be clocked at the same frequency.

If there are multiple clock inputs, it is recommended that those clocks be capable of being clocked at the same frequency.

Justification: This limits the test vector size.

5.7 I/O and Pads

This section contains rules and guidelines for modeling the I/O interfaces of a VC. The goal is to provide a uniform model and substitution mechanics for the behavior.

VSI FV Rule 5.7.1: SoC modeling views must assume default strength values.

The default strength must be the value assumed by the simulator. Do not drive strong logic values on pull up or pull down net. The simulator default strength should be used so that the value can be overridden easily

Justification: This ensures the ability to reuse VC drivers between different SoCs, and between RTL and gate simulations without experiencing contention.

5.8 Messages

This section contains rules and guidelines for how errors, warnings, and informational messages are to be issued during simulation. The goal is to provide a uniform messaging behavior to aid in script writing, readability, and portability.

VSI FV Rule 5.8.1: A common routine must be used to display simulation messages.

Justification: Using common routines to display messages ensures a uniform output format and simplifies debugging and script writing. A single display routine also allows a single point of maintenance for the log file names.

VSI FV Rule 5.8.2: A common format must be used for reporting warnings and errors during the simulation.

The format for reporting informational messages during the operation of a verification environment uses the following form:

```
<sim_time>:<message_source>:<severity>:<message>
```

Where:

<sim_time> is the simulation time at the trigger point for the message

<message_source> is the code module or body that the message originates from

<severity> is either *Info*, *Warning*, or *Error*

<message> is the message body in free-form text

Example:

The string "MY_MODULE" is the point of origin in the following message:

```
780:MY_MODULE:Error:Unexpected Bus Abort
```

VSI FV Rule 5.8.3: Displayed comments should be limited to 80 characters.

All display comments in a stimulus should be limited to 80 characters. The only exceptions to this guideline and for signal scope or directory path names.

Justification: Limiting the comment to 80 characters improves readability by preventing word wrapping.

Exception: The first exception is for displaying the scope of a signal. For example, "`top.testbench.interfacemodel.module1.signalA`" should not be part of the 80-character limit.

Exception: The second exception is for a directory path names. For example, the testbench may display the path to configuration or stimulus files.

VSI FV Rule 5.8.4: Messages should be used instead of embedded comments.

It is recommended that messaging be used to put helpful information into output files instead of using only embedded stimulus comments that are only seen in a source file.

Justification: When debugging, it is useful to have information displayed in the output files to assist with relating errors back to the stimulus.

5.9 Termination

All testbenches must be terminated by a standard mechanism. This ensures that VC integrators can easily determine if the verification passed or failed without any detailed knowledge of the block or the stimulus.

VSI FV Rule 5.9.1: The testbench must complete execution with a pass or fail indication.

Pass and fail must be mutually exclusive. If a test does not pass, it must indicate failure. Time-outs are a failure. *Case* or *if* statements must be complete to indicate if a test passes or fails. There must be no branches that allow the test to finish without indicating pass or fail.

Justification: Parsing scripts may not notice failures.

VSI FV Rule 5.9.2: A passing test must end with a return code of zero if numeric return codes are used.

Justification: Automation scripts can detect failing tests using non-zero return codes. There must be no branches that allow the test to finish without indicating pass or fail.

VSI FV Rule 5.9.3: Simulations must indicate pass or fail at completion with a standard format.

Simulation runs may be comprised of one or more tests. At the completion of a simulation, the final message must indicate precisely whether or not the test passed or failed. One way to do this is via a message and a standard format must be adopted and documented. For example, the following messages could be used (case sensitive):

- "Simulation completed successfully."
  Use this message if there are zero warnings or messages.
- "Simulation completed with errors!"
  Use this message if is one or more errors and zero or more warnings.

- “Simulation completed with warnings!”
  Use this message if there are one or more warnings and zero errors.

VSI FV Rule 5.9.4: Errors must be indicated by common routines or mechanisms.

Errors must be indicated to the testbench using an error signal interface or call to a centralized error-handling task. The error-handling task must use the standard message routines (see the previous rules). Drivers, monitors, and behavior checkers must flag errors using a centralized messaging- and error-handling mechanism.

> **Justification**: A standard mechanism for handling errors simplifies testbenches and increases the portability of drivers.
>
> **Example**: Unimplemented commands, erroneous tests, and miscompares of actual expected results.

VSI FV Rule 5.9.5: Keywords must not be used in the stimulus.

Keywords such as Error and Warning must not be printed as part of a regular display comment. Special tasks must be used to indicate warnings and errors. A complete set of keywords is available from the design and verification environment.

> **Justification**: Improper use of keywords may confuse scripts, resulting in incorrect pass or fail status.

VSI FV Rule 5.9.6: The testbench must terminate with a single statement.

A single statement in the testbench must be the source of the call that terminates the simulation.

> **Justification**: This centralizes maintenance on one block of code for termination and Error or Warning message logging. It also allows a central location to delay termination if additional context is required.

VSI FV Rule 5.9.7: Hang detection must be provided.

Hang detection must be provided for features that wait for events to occur. If an event never occurs, the stimulus must finish. To guarantee that stimulus does not hang during these scenarios, hang-detection code must be provided.

> **Justification**: All tests must terminate. In the event of a deadlock situation (for example, a feedback loop required for stimulus continuation does not occur), a means must be available to terminate out of this section of stimulus. This should include either terminating the test case, or advancing to the next section of the test case. A message must also be provided to indicate that the action occurred.

5.10 Synchronization

Production test patterns must maintain synchronization with the tester. Since the verification testbench is often used to generate some of the production test vectors, following a few rules will greatly ease the translation process. It is critical to change external pin values with respect to a clock edge, and to include a means of resetting free-running counters to guarantee a predictable count value.

VSI FV Rule 5.10.1: It must be possible to set counters by using test stimulus.

Free running counters must be clearable in normal mode or a special mode, so that the count is predictable with respect to some system level input.

> **Justification**: The outcome of stimulus does not depend on the prior state of the SoC when stimulus are executed singularly and then concatenated.
>
> **Example**: If a clock divider is free running with respect to the system reset, only the first stimulus on a tester correctly predicts the events generated by the counter in the time domain of the divided clock.

VSI FV Rule 5.10.2: Synchronous signals must be driven synchronous to an edge transition.

When modeling synchronous waveforms, synchronization must be included in the generation of the dependent signal.

Justification: If arbitrary delays are used, it is easy for signals to move out of synchronization with respect to delays in the clock.

Example: Instead of using hard time delays to synchronize, it is better to use the edge of the signal as the trigger to generate the dependent signal.

VSI FV Rule 5.10.3: Signals used to derive other signals must be derived themselves.

The base signal must never be used by processes outside of its current simulator process.

Justification: Use of derived signals may cause unintended skew between a base signal and derived signal while the simulator is evaluating the derived signal. If a base signal is re-derived in line with the signal derived from it, both are guaranteed to be synchronous independent of the simulator evaluation order.

Example: If using clk to generate clk2, clk must be regenerated within the process (along with clk2). After that, the regenerated clk and clk2 can be outputted.

VSI FV Rule 5.10.4: Synchronous data signals must be driven with a delay between the clock and data output.

Justification: This avoids timing issues within the simulator.

5.11 External Interface Functions (Such as PLI Routines)

The following rules standardize the methods for interfacing external programs with a simulator.

VSI FV Rule 5.11.1: C routines must be written so that they can be loaded dynamically by the simulator.

Justification: Common usage makes sharing C programs easier. This also allows dynamic loading of calls from a compiled library. No test-bench compilation is required for new C functions.

5.12 Configuration Control

The following rules and guidelines standardize the methods for setting configurations within verification components such as the testbench, drivers, and monitors. Configuration control is the process by which the modes of a verification component are set. For example, a design block may have a specific boot mode setting that is configured by an external driver. The driver value may be set by reading settings from a configuration file or from stimulus commands.

VSI FV Rule 5.12.1: Configurations must be set by the stimulus or the configuration control file.

Justification: The VC integration needs an easy and consistent mechanism to change the mode being verified.

VSI FV Rule 5.12.2: Verification components must assume a known default configuration.

Verification components must assume a known, well-documented default configuration.

Justification: Sparse data is easier to maintain given that the defaults are documented.

VSI FV Rule 5.12.3: Default configuration settings for the VC and testbench must be identical.

Justification: Stimulus and models are reusable without modification. Design under test can be exercised more easily in all configurations.

5.13 VC Reset

All stimuli need a standard means of resetting and initializing the system. In addition, special consideration must be taken to account for memory-based stimulus. This section addresses these issues.

VSI FV Rule 5.13.1: A standard task must be used to reset the VC.

This includes the standard reset configuration value for the stimulus type. This command may initiate a reset cycle on a bus.

> **Justification**: This allows for portability and code reuse.

VSI FV Rule 5.13.2: A single block in the testbench must be the source of the reset function.

> **Justification**: This centralizes maintenance on one block of code for reset.

VSI FV Rule 5.13.3: Forced internal signals for reset must drive both states.

If it is necessary to force internal signals, both logic 0 and logic 1 states must be simulated.

> **Justification**: This is needed to ensure that the part behaves appropriately in either state.

VSI FV Rule 5.13.4: No assumptions must be made about the reset state.

Do not make any assumptions about how the part comes up out of reset, except the defined specification of reset behavior.

> **Justification**: Reset may vary in different environments. Relying on a default reset state of a VC may cause the stimulus to fail when run during SoC verification because the part may configure the VC in a different manner.

> **Example**: Some VCs have register bits that are configurable by other bits at reset. This becomes an issue if the production stimulus assumes one configuration while the part is configuring the device with a different reset configuration. Such a case causes the stimulus to fail unnecessarily. This scenario could happen if the VC is a memory whose control register bits are determined by a shadow row that is configured by the customer.

5.14 Testbench Interface

This section covers the standards for how the testbench controller interfaces with the drivers and monitors.

The following set of rules and guidelines are common to drivers, monitors, and behavioral models.

VSI FV Rule 5.14.1: Clock inputs must be driven by a top-level clock generation module.

All clocks must be derived from a configurable clocking module.

> **Justification**: This provides consistency and control of main clock domains and frequency.

VSI FV Rule 5.14.2: Communication to drivers and monitors must occur without advancing simulation time.

The test-bench sequencing interface to the driver, monitor, and behavioral interfaces must communicate back and forth in zero simulation time. Use of #0 in Verilog is not allowed, so this statement should be worked around by structuring the code so that it flows with the sequence of events and makes specific handshakes.

> **Justification**: The testbench must be able to pass a transaction to the driver without advancing simulation time.

5.15 Behavioral Models for Memory

This section covers the standards for coding and providing behavioral models for the memories in a VC.

VSI FV Rule 5.15.1: It is recommended that both a detailed and a high-level behavioral model of memory be supplied.

Such models can be used for the following purposes:

- The addition of many value-added features such as integrated coverage metrics and callbacks
- Minimization of simulation memory footprint as compared to an instantiation of the actual memory component
- Acceleration of simulation as compared to an instantiation of the actual memory component.

VSI FV Rule 5.15.2: The dimensions of arrays must be parameterized.

Justification: Parameters for array size and dimension make arrays more portable.

VSI FV Rule 5.15.3: It is recommended that memory models separate control and storage functions.

Any behavioral memory models supplied should be designed to have separate control (for example, memory interface handshaking) and storage functions.

Justification: This allows integrators flexibility and the possibility of replacing the storage functions with another technique while maintaining adherence to the interface.

VSI FV Rule 5.15.4: It is recommended to limit memory sizes to the size required by the stimulus.

Memory space restrictions may be established for modeling purposes and to speed simulation.

Justification: Smaller memory size speeds up simulation.

5.16 Memory Operation

VSI FV Rule 5.16.1: A RAM must not be pre-loaded.

RAM models should leave the memory array uninitialized. A common source of verification errors is an unknown dependence on the contents of a data RAM location. Pre-loading a RAM with a pattern may inject an unintended error into the verification scenario. This rule is not intended to apply to pre-loading of op-code data for software-based tests.

VSI FV Rule 5.16.2: Memory arrays should be modeled using a standard simulator API or optimized models.

Memories that are used internally to the VC under test, or are used in the VC function verification environment should be modeled efficiently. This implies the use of a standard simulator API (for example, VPLI for Verilog simulators and FLI for VHDL simulators), or optimized models that are written by the VC provider or are available commercially.

If a standard API is used, the code supplied should adhere to the standard implementation of that API.

Justification: This increases the execution speed and flexibility.

Example: A Verilog example is:

The call `$read_32bit_data (reg data[31:0], wire addr[31:0])` returns in the register *data*, a 32-bit, word-aligned value determined by the word-aligned address, *addr*.

The call `$write_32bit_data(reg data[31:0], wire addr[31:0])` places the 32-bit, word-aligned *data* into the PLI memory at the word-aligned address, *addr*.

Note that byte writes can be implemented by using a `$read` followed by a `$write` after the appropriate bit manipulations have been completed.

VSI FV Rule 5.16.3: Stimulus binaries must be parsed and loaded into memory using standard functions.

A standard set of routines must be used to enable parsing of compiled stimulus binaries, to enable the upload into high-level memory arrays, and to enable high-level array modules at the SoC level as well as memory models at the test-bench level.

> **Justification**: This increases the execution speed and flexibility. Stimulus is not bound by a statically compiled array size or the need to fracture a stimulus into multiple files for `readmemh` processing.

> **Example**: A stimulus that defines constant data for a high level ROM model in an SoC uses a `$read_32bit_data(reg data[31:0],reg addr[31:0])` call to replace the array. The stimulus is linked to locate the necessary data within the ROM address range. The high level array relies on the SoC memory module control blocks to generate the address attributes. The high level ROM array appends the constant address bits to the generated address bits and returns a 32-bit data value. This data value is then aligned as needed and placed into the data path.

5.17 Detailed Behavioral Models for I/O Pads

This section covers the coding and naming of I/O pad models.

VSI FV Rule 5.17.1: The pad model must be named <module>_pad.<extension>

> **Justification**: This ensures easy identification in the deliverables directory structure.

5.18 Stimulus

These are the inputs to the testbench that stimulate the VC within the testbench. This section outlines the rules that apply to stimulus, regardless of the stimulus form.

5.18.1 Random

Random simulation is used for exercising boundary cases of the VC. It is achieved by generating pseudo-random transactions for stimulus. The randomness can be either be in content (such as random data in a write transaction) or in sequence (for example, randomly choose between a read and a write transaction).

Constraints are written to specify the range of allowed random transactions. Tools are used to randomly generate transaction within the allowed range. Probability and weighting schemes are used to bias the random selection of transactions.

This section covers the pseudo-random testbench standards. It includes rules and guidelines for random generation. However, the pseudo-random testbench standards are not mandatory, and the following rules and guidelines are applicable only if random simulation is done.

VSI FV Rule 5.18.1: Random stimulus must include a checking mechanism.

A reference model or a prediction function must be provided to verify the random behavior of the VC.

> **Justification**: As opposed to non-random testbenches where expected results can be embedded within the stimulus, random behaviors need a more general way to describe the expected results.

VSI FV Rule 5.18.2: All random stimuli must be able to be captured and delivered as standalone tests.

> **Justification**: The VC integrator must be able to run statically saved random tests without the random generation function.

5.19 Checking

Stimulus must provide an automated mechanism to determine pass or fail. The verification stimulus must be self-contained so that the results are not subject to interpretation.

VSI FV Rule 5.19.1: The checking mechanism used must be automated.

Justification: This eliminates potential for errors, or misinterpretation of results.

VSI FV Rule 5.19.2: All stimuli must be self-checking.

Stimulus must provide a means for automated self-checking. When stimulus fails, an error must be indicated, informing the user that the test or regression has failed. All stimuli must rely on a detailed behavioral model so that the VC can check results during run time, or else the stimulus must include checking for expected results.

Justification: This provides a way for automatic checking of verification stimulus results and eliminates the potential for errors.

VSI FV Rule 5.19.3: Errors must be detected at the point of failure.

As often as possible, identification of the failures must be at the point of failure, and not post-processed.

Justification: This reduces the time spent debugging failures.

VSI FV Rule 5.19.4: Simulator errors and warnings must be detected.

Post-processing scripts must take into account simulator messages (for example, compilation, invocation errors, and so on).

Justification: This ensures that the stimulus was properly executed.

5.20 Partitioning

Partitioning can impact the ease with which a model can be adapted to an application. Patterns must be partitioned to facilitate portability to different chips. Proper partitioning allows the easy omission of patterns whose functions or pins are not being utilized on a particular chip. Patterns can be used “as is” without modifications. This directly reduces stimulus debug time.

VSI FV Rule 5.20.1: Any stimulus that depends on another VC must be partitioned.

Any stimulus that depends on another VC must be partitioned. Any accesses to functions outside of a VC must be done with a macro or task.

Justification: On future products, the dependent VC may not exist. A properly partitioned stimulus suite allows for the easy identification and porting of patterns that specifically depend on that module.

VSI FV Rule 5.20.2: Module patterns must be partitioned based on the mode of operation and functionality.

The patterns must be in individual files for the regression runs. The stimulus file naming must follow documented naming conventions.

Justification: Having modular and independent stimulus allows faster identification of problems in the failing tests.

VSI FV Rule 5.20.3: Timing-critical patterns must be partitioned.

Any patterns that verify a timing-critical path must be partitioned into a separate stimulus.

Justification: In future products, the timing path may not exist due to re-synthesis. In addition, timing critical patterns are typically part dependent, and careful consideration must be taken when porting them to another product.

VSI FV Rule 5.20.4: Test-mode patterns must be partitioned.

Any stimulus that requires a test mode must be partitioned into a separate stimulus. Test mode is an operating configuration that allows hardware to be tested more easily than it would be in normal operating conditions. These tests may be run in master mode, slave mode, and so on. They should follow a documented naming convention. The tests should be partitioned separately from the rest of the test stimulus, since a special mode of operation is required.

Justification: Typically, a test mode is different on different products; therefore careful consideration must be taken when porting test-mode patterns to another part.

VSI FV Rule 5.20.5: Stimulus must be partitioned into startup, body and shutdown sections.

Justification: This fosters reuse of body code. Start-up and shutdown code may be different when running at the system, SoC, or module level. If the stimulus is partitioned, reusability is enhanced.

VSI FV Rule 5.20.6: Module patterns should be partitioned based upon functionality

Individual patterns should be partitioned into separate files based upon functional test cases. Partitioning should be done to avoid redundancy and ensure proper ordering of tests.

Justification: A test pattern that checks a small number of test cases allows faster identification of problems in the failing stimulus and easier generation of test vectors.

Example: The test order should exercise the read and write capability of a register prior to testing the function of a register.

5.21 Naming

The following section contains rules and guidelines to specify naming and conventions for verification stimulus files and directories.

VSI FV Rule 5.21.1: Stimulus must indicate the stimulus version at run time.

At run time, stimulus binaries must display a revision control number that identifies the version of the source code.

Justification: To allow identification of a stimulus version.

VSI FV Rule 5.21.2: Stimulus source files must be named using a documented naming convention.

Justification: This improves readability and makes it easy to identify what module and mode of operation the stimulus is assigned to.

Example: For slave mode, “slave” must be used in the mode field and “intmem” and “extmem” must be used for master mode patterns. “Emul” must be used for emulation mode. New modes may be created but must be documented in the verification guide. More than one field is allowed for the description; however, the description must reside after the mode field.

5.22 Memory Map Control

This section outlines the rules necessary to build an interface wrapper to the external memory models and peripherals at the unit stimulus level.

VSI FV Rule 5.22.1: The configuration file must include control statements for memory.

The configuration file contains mode settings for testbench components.

Justification: Provides flexibility to verify any memory configurations.

Example: For example, wait states is an asynchronous aspect of memory that must be parameterized using the configuration file.

VSI FV Rule 5.22.2: The configuration file must define wait states for external SoC memory ranges.

Justification: It is necessary to regulate memory map wait state arrangements as well as to model sequential memory banks with differing wait-state configurations.

Example: The address range 0x4-0x19 could be assigned a wait state of 5 cycles, while the address range 0x1a-0x200 is assigned a wait state of 3 cycles. Note that support is at the byte level.

VSI FV Rule 5.22.3: The default wait-state configuration must be the minimum supported configuration.

Justification: This increases simulation speed.

VSI FV Rule 5.22.4: Memory interface wrappers must be dynamically configurable.

Justification: This allows the ability to run multiple memory configurations within the same stimulus.

VSI FV Rule 5.22.5: Misaligned accesses should be tested in master mode.

Example: Testing alignment in master mode makes the tests portable to other systems. Different buses may handle misalignment differently, causing tests written in slave mode to be non-portable.

5.23 Stimulus Source Code

Properly coded patterns can produce patterns that are portable and easier to maintain. The following standards and guidelines are for verification source code.

VSI FV Rule 5.23.1: The information messages generated by verification code should be sufficient to enable understanding of the code.

The information messages in verification code should allow an end user to understand the error that has occurred and if necessary to trace the error back to the relevant line in the verification source code where it is provided. When a pre-compiled verification file is provided, the informational messages are all that the end user has to work with. In this case, the content of the messages is critical to making a diagnosis of the source of an error.

VSI FV Rule 5.23.2: The stimulus source code must document the features it verifies.

The stimulus source code must contain comments, in addition to the header, within the context of the stimulus documenting the tests that occur in the code flow. Functional stimulus patterns should be well commented or otherwise self-documenting so that the pattern source clearly indicates the function being tested. This enables the failure to be related back to the location of the defect in RTL.

Justification: This provides the means by which the VC consumer can relate a functional pattern failure back to the design feature that has the defect.

VSI FV Rule 5.23.3: The header documentation must match the stimulus.

Justification: This ensures that proper test strategies and verification methodologies are being followed.

VSI FV Rule 5.23.4: The header must follow a documented contents definition.

Justification: This enables automatic parsing and improves readability.

Example: See the example content shown in the following figure.

```
// +FHDR----------------------------------------------------------------------
// Optional Copyright {c}
// Optional Company Confidential
// ---------------------------------------------------------------------------
// FILE NAME      :
// DEPARTMENT     :
// AUTHOR         :
// AUTHOR'S EMAIL :
// ---------------------------------------------------------------------------
// RELEASE HISTORY
// VERSION DATE       AUTHOR DESCRIPTION
// 1.0     YYYY-MM-DD name
// ---------------------------------------------------------------------------
// KEYWORDS: General file searching keywords, leave blank if none.
// ---------------------------------------------------------------------------
// PURPOSE: Short description of functionality
// ---------------------------------------------------------------------------
// PARAMETERS
// PARAM NAME RANGE:DESCRIPTION:DEFAULT:UNITS
//
// ---------------------------------------------------------------------------
// REUSE ISSUES
//   External Pins Required:
//   Monitors Required:
//   Drivers Required:
//   Local Functions:
//   Include Files:
//   Other:
// ---------------------------------------------------------------------------
// FEATURES TESTED:
// ---------------------------------------------------------------------------
// DETAILED TEST DESCRIPTION:
// -FHDR----------------------------------------------------------------------
```

Figure 4: Example of VSI FV Rule 5.25.[illegible]

A. Glossary

Term	Definition
Application-Specific Prototype	A prototype design built from commercially available components
Assertion monitors	Monitors to check that properties of the VC hold during dynamic simulation; also called assertion checkers
Assertion or property	A statement of design intent that can be checked in dynamic simulation and formal verification
Behavioral Model	Model that exhibits some or all of the functionality of the artifact being modeled but is not written to be taken through a design flow and therefore may not be synthesizable)
Boolean or Structural Equivalence Checking	A formal equivalency check in which pairings of inputs, outputs and memory elements are first determined for each of two design versions, and then the combinational cone of logic at the inputs to each memory element or output of a pair is proven equivalent, meaning it is proven that the same truth table is implemented
Branch Coverage	Measures which branches were executed, for example, “case” or “if...else” branches
Bus functional model or BFM	Used to provide simplified bus agent models for verifying VCs that attach to buses (such as the AMBA bus)
Code Coverage	Coverage metrics defined in terms of syntax of the design and measured during dynamic simulation of the design, including statement coverage, toggling of variables, finite state machine transition and state visitation coverage, if-else branch coverage, conditional statement coverage (metrics on the ways that a condition can become true), paths through if-else and case statements and general signal coverage
Compliance tests	Tests provided to demonstrate that a VC complies to some agreed standard (such as the AMBA bus specification protocol)
Constraint	Rules defining relationships between signals within a design; they can be combinatorial or sequential/temporal and can be used in pseudo-random generation and formal methods, for example
Coverage monitors	Monitors that checks that certain events occur in the VC during dynamic simulation
Cycle-Based Simulation	A simulation in which each element of a design is evaluated only once per clock cycle; can be contrasted to event-based simulation
Directed or Deterministic Simulation	Simulation in which the stimulus is specified explicitly, as is the expected response of the design model; can be contrasted with random or pseudo-random simulation
Driver	The part of the testbench that drives values onto the signals of the VC being verified; it is considered good testbench design style that they only drive interface signal

Term	Definition
Dynamic Formal Verification	typically exploring only a portion of the state space and therefore potentially less exhaustive than model checking from the reset stat
Dynamic Verification	Execution of a model or models of a design with a set of stimulus
Emulation	Specially designed hardware and software systems, using some re-configurable hardware, such as FPGAs, that can simulate a hardware design faster than conventional workstation or PC-based simulators
Equivalence Verification or Checking	Process of determining whether two designs (which could be of differing levels of abstraction or format) match in terms of functionality; equivalence checking is often performed statically using formal methods
Event-Based Simulation	A simulation in which events (changes of input valuations, which may occur multiple times during a clock cycle) are propagated through a design until a steady state condition results; can be contrasted to cycle-based simulation.
Expected Results Checkers	A means for checking the results of a simulation against a previously specified, correct response
Expression Coverage	Measures how well a boolean expression has been exercised, for example, the boolean expression used in an "if" condition
Formal Constraint-Driven Stimulus Generation	The use of formal methods to generate targeted tests that satisfy a set of constraints
Formal Coverage or Semi-Formal Verification	The use of static or dynamic formal methods to improve coverage results as measured by some appropriate metric
Formal Equivalence Checking	The application of formal methods to equivalence verification; Boolean/structural and sequential equivalence are two examples of formal equivalence checking
Formal Verification	The use of mathematical techniques and formalisms to verify aspects of a design, spanning both intent and equivalence verification; such techniques are often called static because they do not involve execution of the design and can be contrasted to dynamic verification
FSM Arc Coverage	Shows how many transitions of a Finite State Machine (FSM) were executed
Functional Coverage	Coverage metrics, generally related to behavior that changes over time and sequences of events such as tracking of bus interactions; these need to be individually defined by people knowledgeable about the design and its intent
Functional to RTL Test Suite Migration	A means for translating a test suite for an abstract behavioral model of a design into a test suite suitable for the RTL level
Golden Model Checkers	Simulation monitors that check the responses of two models of a design, one of which is considered the reference or "golden" model
Hardware Acceleration	A system for mapping all components of a software simulation onto a hardware platform that is specifically designed to speed up the simulation process
Hardware Modeling	A system in which a simulator receives input from and sends output to a hardware component
Hardware/Software Co-Verification	A system in which the hardware and the software portions of a design are executed and verified in parallel
Input constraint	A constraint on input signals

Term	Definition
Integration Verification	contains one or more virtual components
Intent Verification	Process of determining whether a design fulfills a specification of its behavior
Model Checking or Property Checking	A formal verification technique for checking the entire state space of a design for violations of properties, for example, specifications of behavior
Monitor	Monitors are probes that observes signals in the VC during dynamic simulation of the VC
Path Coverage	Shows which routes through sequential "if...else" and "case" constructs have been exercised
Physical Prototyping	A hardware representation of a design (often created using FPGAs) that operates at speeds close to, but not necessarily as fast as, the ultimate design to be built
Physical Verification	The process of checking the geometric design database to ensure that the physical implementation is a correct representation of the original logic design, consisting of three distinct checks: Electrical Rules Checks (ERC), Design Rules Checks (DRC) and Layout Versus Schematic Checks (LVS)
Protocol Checkers	A means for checking behavior of an interface and determining if violations of defined, acceptable behavior have occurred
Provider Functional Verification	Verification performed by the VC provider
Pseudo-random Simulation	A dynamic simulation technique where the design is stimulated with pseudo-random inputs by the user exercising some control over the random stimulus generation; can be contrasted with directed and random simulation
Random or Non-Directed Simulation	A dynamic simulation technique where the design is stimulated with random inputs; can be contrasted to directed and pseudo-random simulation.
Reconfigurable Prototyping System	A system in which the VCs of an SoC design are created in off-the-shelf components, bonded-out silicon, FPGAs or in-circuit emulator systems
Register Transfer Language (RTL)	A programming language representation of a design in which some, but not all of the design structure is explicitly represented, such as the Verilog and VHDL languages
Regression Testing	Techniques for running large numbers of "verifications" (such as tests and property checks) in batch mode, with minimal human intervention, with results analyzed in batch mode and pass/fail outcomes reported in an automatic way
RTL to Netlist Test Suite Migration	A means for translating a test suite that operated on the RTL level to one that operates on the netlist level of a design
Sequential Equivalence Checking	Formal equivalency checking techniques that require mapping of inputs and outputs but do not rely on mapping of memory elements in one design to another, but rather prove, given a set of initial states for each of the designs, that designs with different numbers of, or different arrangements of, memory elements produce the same output streams given the same input streams
Signal Coverage	Shows how well state signals have been exercised
Statement Coverage	Shows how many times a statement in the RTL was executed

Term	Definition
Static Functional Verification	of verification test suites; there is no industry consensus on the verification approaches included under this term
Stub model	A particular type of behavioral model that only models the interface signals to allow connectivity to be tested. Outputs may be assigned values in the stub model
Symbolic Simulation	Simulation in which some or all inputs are symbolic variables, and functions of these variables are propagated through a design
System-on-Chip (SoC)	A single piece of silicon containing multiple VCs to perform a certain defined function
Testbench	The overall system for applying stimulus to a design and monitoring the design for correct responses and functional coverage
Theorem Proving	A formal verification technique in which a specification is expressed in a formal logic and proof strategies are utilized to construct a proof that a design obeys the specification
Toggle Coverage	Shows which bits of the signals in the design have toggled
Triggering Coverage	Shows whether each process has been uniquely triggered by each of the signals in its sensitivity list
VC Verification	Process of verifying the functionality of a virtual component, for example, unit test of that component
Verification Metrics	Techniques for measuring the effectiveness of verification procedures on a design; these include code coverage metrics, functional coverage metrics and bug-tracking metrics
Verification Test Suite Migration	A means for translating a test suite that operated on one design level (for example, gate netlist) to another level such as RTL
Virtual Prototyping	A simulation model of a component or an entire system, useful for exploring design alternatives and testing for correctness
Visited State Coverage	Shows how many states of a Finite State Machine (FSM) were entered during simulation